



Techniques for Debugging HPC Applications

NIKOLAY PISKUN , TOTALVIEW SOFTWARE ARCHITECT

AUGUST 11, 2021, ATPESC 2021

Agenda

- Introduction
- Overview of TotalView Features
- TotalView Debugging Solution
- General Debugging Features for C, C++, and Fortran
 - UI Navigation and Process Control
 - Action Points
 - Examining and Editing Data
 - Advanced C++ and Data Debugging
- Mixed Language C/C++/Fortran and Python Debugging
- Remote Debugging
- MPI, OpenMP, CUDA GPU, and Hybrid Debugging
- Reverse Connect/Attach
- Memory Debugging
- Reverse Debugging
- HPC Debugging Techniques
- TotalView Resources and Documentation
- Q&A

What is Debugging and
Why do you need TotalView?

TotalView Features

- Comprehensive C, C++ and Fortran debugger
- Multi-process/multi-thread dynamic analysis
 - Thread specific breakpoints with individual thread control
 - View thread specific stack and data
 - View complex data types easily
- MPI, OpenMP, Hybrid and CUDA debugging
- Convenient remote debugging for HPC
- Integrated Reverse and Memory debugging
- Mixed Language - Python C/C++ debugging
- Script debugging
- Linux, macOS and UNIX

The screenshot displays the TotalView 2020 debugger interface for a multi-threaded application. The main window shows the source code of a worker thread, with a breakpoint set at line 49. The left sidebar shows the process and thread hierarchy, including a list of threads and their states (Running, Stopped). The right sidebar shows the call stack and local variables for the current thread. The bottom status bar indicates the current process, thread, frame, file, and line.

```
void Worker::doWork()
{
    qDebug() << "Starting worker process in Thread " << thread()->currentThreadId();
    qsrand(worker);
    do {
        mutex.lock();
        bool doAbort = abort;
        mutex.unlock();
    } while (doAbort);

    switch (worker) {
    case 1:
        qDebug() << "Worker 1 doing work!";
        break;
    case 2:
        qDebug() << "Worker 1 doing work!";
        break;
    case 3:
        qDebug() << "Worker 1 doing work!";
        break;
    case 4:
        qDebug() << "Worker 1 doing work!";
        break;
    }
}
```

ID	Type	Stop	Location	Line
1	Eval	N/A	.../worker.cpp#49	worker.cpp (line 49)

Name	Type	Value
Arguments		
this	Worker * const	0x022241a0 -> (Worker)
Block at Line 45		
doAbort	bool	false (0)
loop	QEventLoop	(Opaque QEventLoop)

What is TotalView used for?

- **More than just a tool to find bugs**
 - Understand complex code
 - Improve code quality
 - Collaborate with team members to resolve issues faster
 - Shorten development time
- Finds problems and bugs in applications including:
 - Program crash or incorrect behavior
 - Data issues
 - Application memory leaks and errors
 - Communication problems between processes and threads
 - CUDA application analysis and debugging
 - Applications in an automated test and batch environments



UI Navigation and Process Control

TotalView's Default Views

1. Processes & Threads

Control View

- Lookup File or Function
- Documents

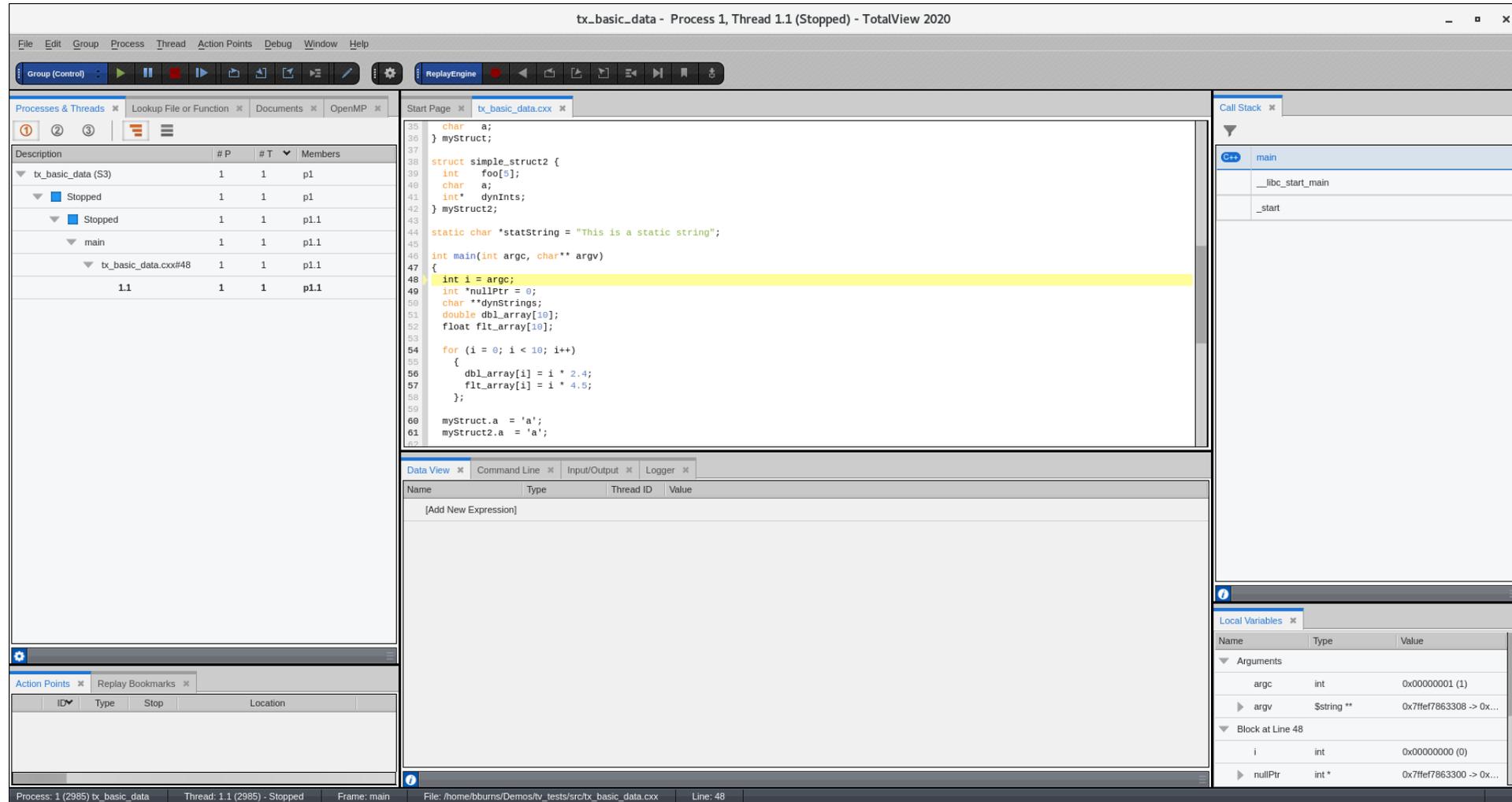
2. Source View

3. Call Stack View

4. Local Variables View

5. Data View, Command Line, Input/Output

6. Action Points, Replay Bookmarks



Process and Threads View

Description	# P	# T	Members
tx_fork_loop (S3)	4	4	p1-4
Breakpoint	4	4	p1-4
Breakpoint	4	4	p2.1, p4.1, p3.2,...
snore	4	4	p2.1, p4.1, p3.2,...
tx_fork_loop.cxx#682	4	4	p2.1, p4.1, p3.2,...
1.3	1	1	p1.3
2.1	1	1	p2.1
3.2	1	1	p3.2
4.1	1	1	p4.1
Stopped	4	8	p1.1, p3.1, p1-2-...
__select_nocancel	2	3	p1-2.2, p2.3
<unknown line>	2	3	p1-2.2, p2.3
1.2	1	1	p1.2
2.2	1	1	p2.2
2.3	1	1	p2.3
snore	3	5	p1.1, p3.1, p4.2,...
tx_fork_loop.cxx#682	3	5	p1.1, p3.1, p4.2,...
1.1	1	1	p1.1
3.1	1	1	p3.1
3.3	1	1	p3.3

Select process or thread attributes to group by:

- Control Group
- Share Group
- Hostname
- Process State
- Thread State
- Function
- Source Line
- PC
- Action Point ID
- Stop Reason
- Process ID
- Thread ID
- Process Held
- Thread Held
- Replay Mode

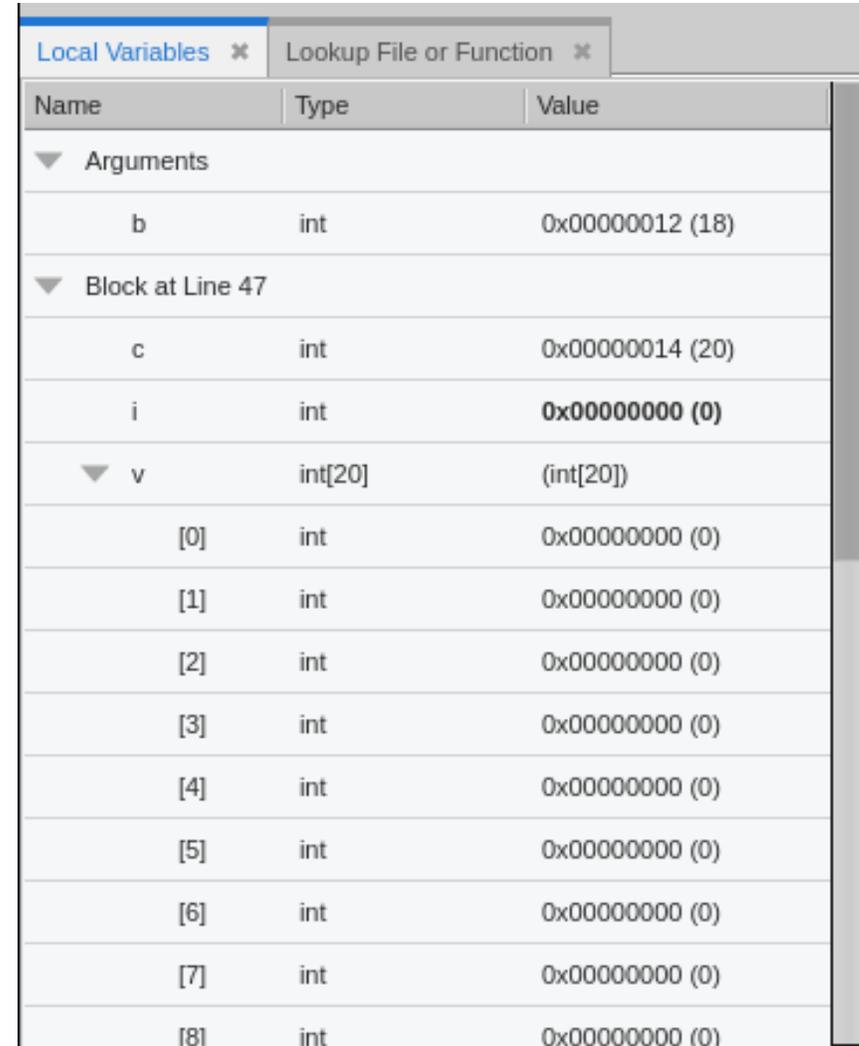
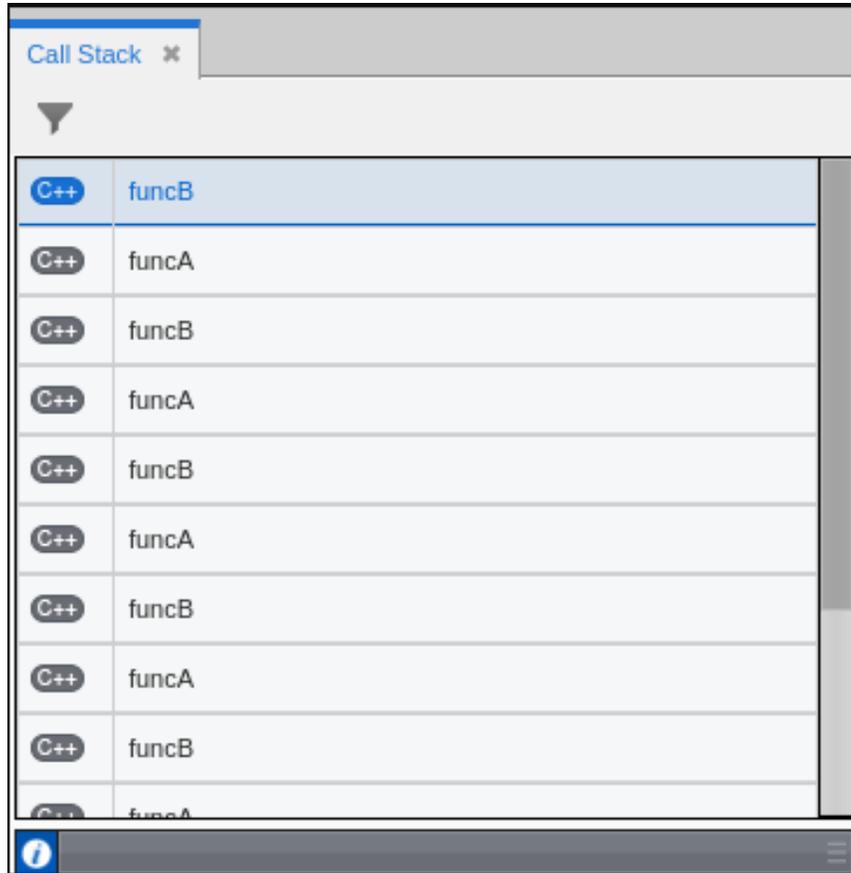
↑ ↻ ↓

Source View

```
Start Page * tx_fork_loop.cxx *
1088 #else
1089     whoops = pthread_create (&new_tid, &attr, (void*)(*)(void*))forker, (void *)local_fork_count);
1090     else
1091     whoops = pthread_create (&new_tid, NULL, (void*)(*)(void*))forker, (void *)local_fork_count);
1092 #endif
1093     if (whoops)
1094     {
1095         printf("pthread_create failed; result=%d, errno=%d\n", whoops, errno);
1096         exit(1);
1097     }
1098     thread_ptids[total_threads++] = new_tid;
1099     printf ("%d: Spun off %ld.\n", (int)(getpid()), (long)new_tid);
1100
1101     forker (fork_count);                /* Never returns */
1102 } /* fork_wrapper */
1103
1104 /*****/
1105
1106 int main (int argc, char **argv)
1107 {
1108     int fork_count = 0;
1109     int args_ok = 1;
1110     int arg_count = 1;
1111     char *arg;
1112     pthread_mutexattr_t mattr;
1113
1114     signal (SIGFPE, sig_fpe_handler);
1115     signal (SIGHUP, (void*)(int))sig_hup_handler);
1116
1117 #ifndef __linux
1118     /* The linux implementation of pthreads uses these signals, so we'd better not */
```

Find: main | Aa | "w" | 2 matches x

Call Stack View and Local Variables View



Action Points View

OpenMP * Action Points * Data View * Replay Bookmarks * Command Line * Input/Output *						
	ID	Type	Stop	Location	Line	Function
<input checked="" type="checkbox"/>	1	Break	Process	.../ReplayEngine_demo.cxx#27	ReplayEngine_demo.cxx (line 27)	main
<input checked="" type="checkbox"/>	2	Watch	Group	4 bytes @ 0x601058 (arraylength)		

Data View, Command Line View and Input/Output View

The screenshot displays the TotalView IDE interface with three overlapping windows:

- Data View:** Shows a table of variables:

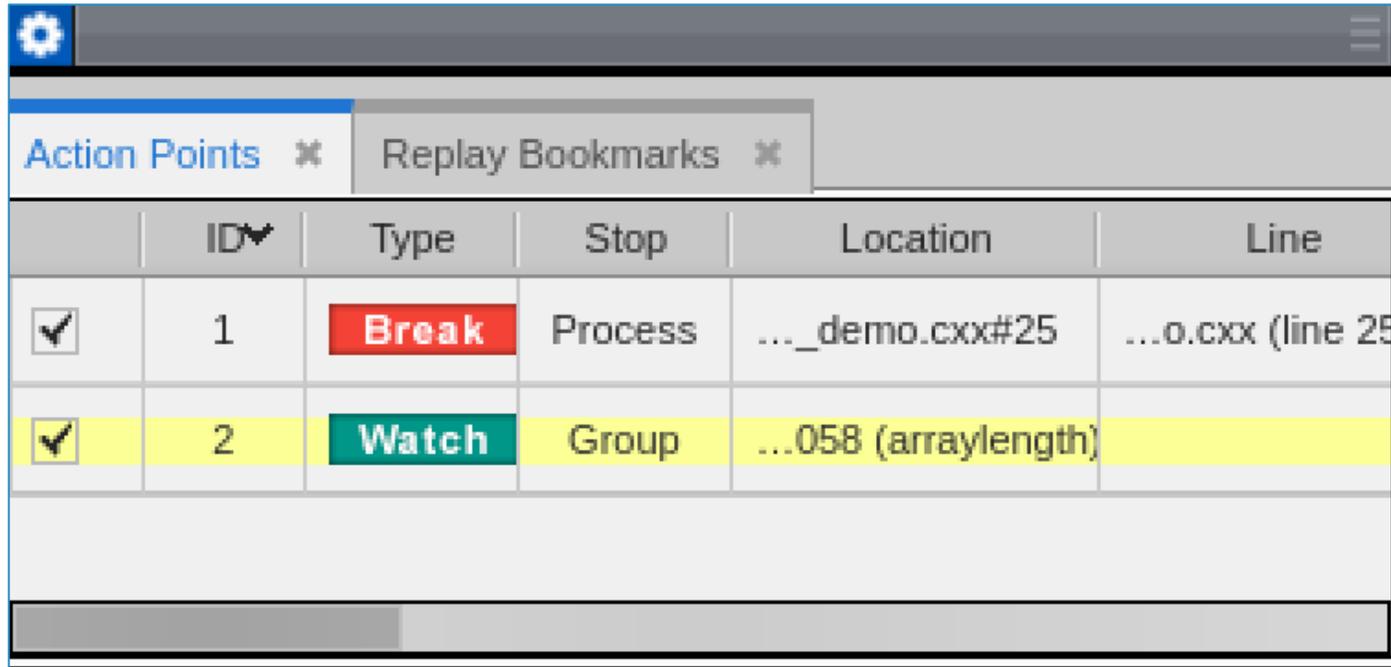
Name	Type	Thread ID	Value
c	int	1.1	0x00000014 (20)
p	int *	1.1	0x7ffe8d6bac60 -> 0x00000014 (20)
*(p)			
v			
[0]			
[1]			
[2]			
[3]			
[4]			
- Command Line:** Shows the execution log:

```
Linux x86_64 TotalView 2020.3.11
Created process 1 (3893), named "ReplayEngine_demo"
Thread 1.1 has appeared
Thread 1.1 hit breakpoint 1 at line 27 in "main"
Thread 1.1 re
Thread 1.1 hi
Thread 1.1 hi
d1.<> dprint
c = 0x00000014
d1.<>
```
- Input/Output:** Shows the output of the program:

```
Flip this text
FLIP THIS TEXT
Flip some more text
FLIP SOME MORE TEXT
```

At the bottom, there is a **Standard Input:** field with a cursor and a **SEND** button.

Action Points



	ID	Type	Stop	Location	Line
<input checked="" type="checkbox"/>	1	Break	Process	..._demo.cxx#25	...o.cxx (line 25)
<input checked="" type="checkbox"/>	2	Watch	Group	...058 (arraylength)	...

Breakpoint

Evaluation Point (Evalpoint)

Watchpoint

Barrier point

Setting Breakpoints

The screenshot shows the TotalView IDE interface. The source code editor displays the following code:

```
30 float b;
31 nestedStruct ns;
32 int nsa[3];
33 };
34
35 int mySub (int x)
36 {
37     x = x*2;
38     return x;
39 }
40
41 int main(int argc, char** argv)
42 {
43     int i;
44     int k = 0;
45     structData myStrucArray[10];
46     int myArray[10];
47     structData myStruct;
48     myStruct.x = 10;
49     myStruct.y = 20;
50     myStruct.b = 777.2;
51     myStruct.ns.a = 2;
52     myStruct.ns.b = false;
53     for (i = 0; i < 3; i++)
54         myStruct.nsa[i] = i*2;
55     const char *charPtr = "This is a string";
56
57     for (i =
58     {
59         int j;
60         int k;
61         myArr
62         myStr
63         myStr
64         myStr
65         myStr
66         print
67         if (
```

A callout box on the right contains the following text:

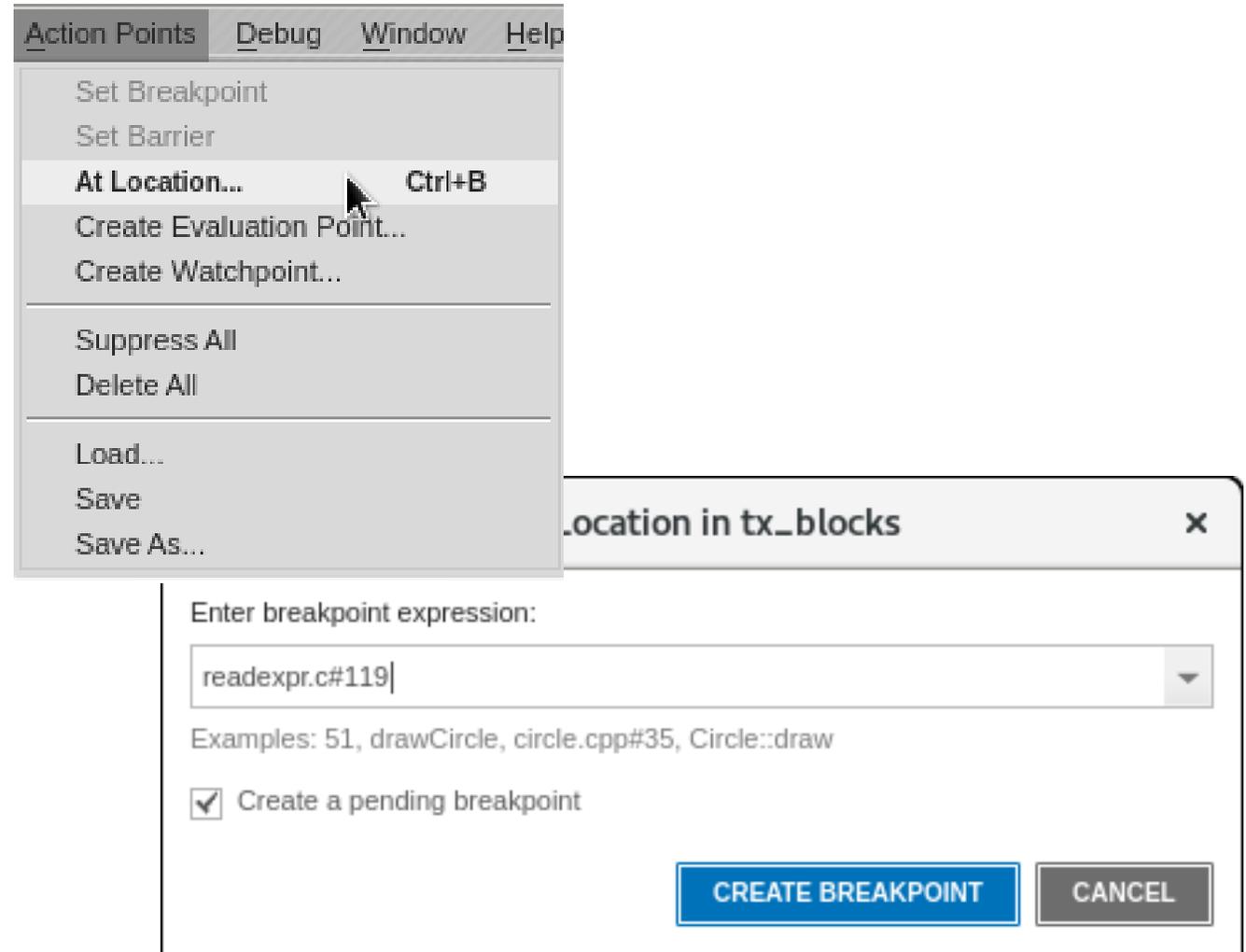
- Setting action points
 - Single-click line number
 - Right clicking on the line number and using the context menu
 - Clicking a line in the source view then selecting the Action Points -> Set breakpoint menu option

The bottom panel shows the Action Points table:

	ID	Type	Stop	Location	Line	Function
<input checked="" type="checkbox"/>	1	Break	Process	.../tx_blocks.cxx#48	tx_blocks.cxx (line 48)	main
<input checked="" type="checkbox"/>	2	Break	Process	.../tx_blocks.cxx#54	tx_blocks.cxx (line 54)	main
<input checked="" type="checkbox"/>	3	Break	Process	.../tx_blocks.cxx#59	tx_blocks.cxx (line 59)	main

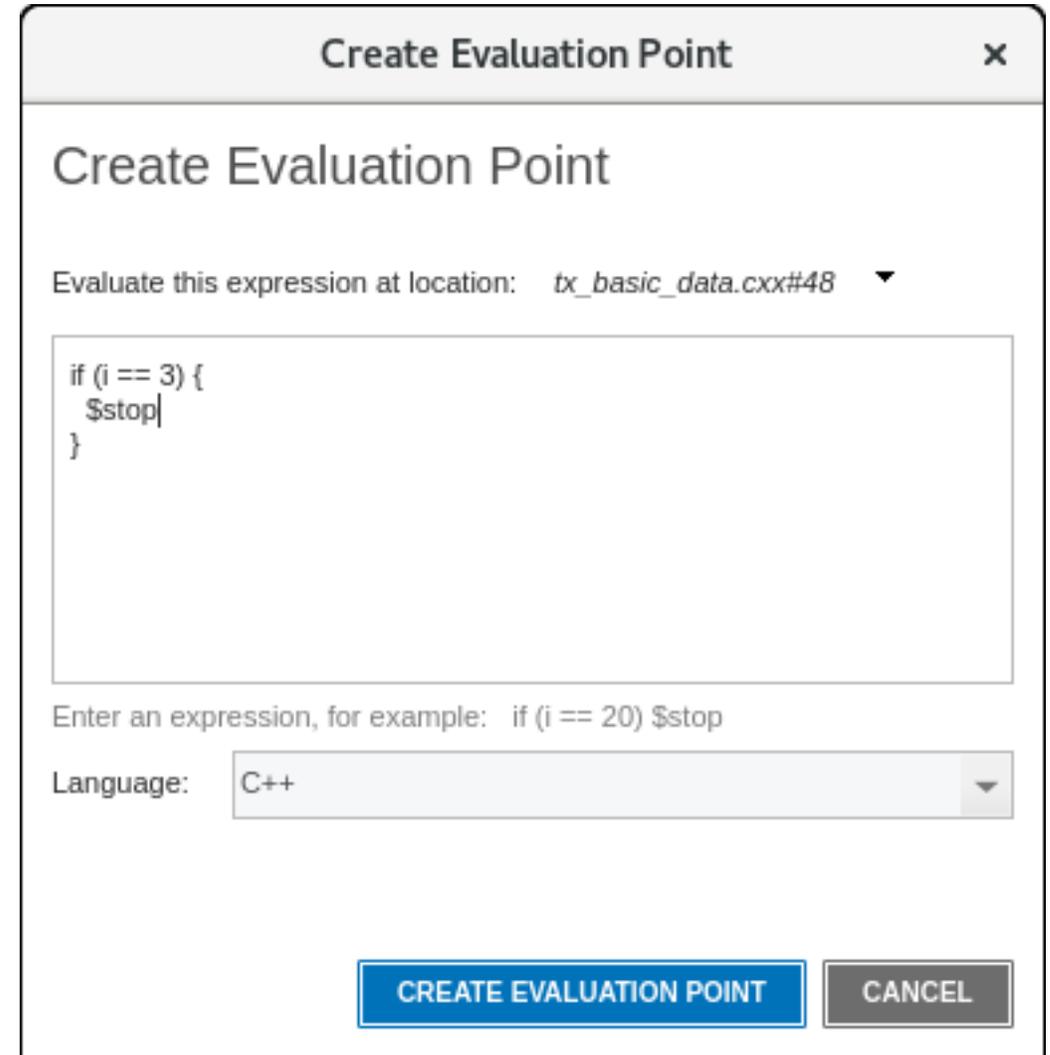
Setting Breakpoints

- Breakpoint->At Location...
 - Specify function name or line number
 - If function name, TotalView sets a breakpoint at first executable line in the function



Evaluation points

- Use Eval points to :
 - Include instructions that stop a process and its relatives
 - Test potential fixes or patches for your program
 - Include a goto for C or Fortran that transfers control to a line number in your program
 - Execute a TotalView function
 - Set the values of your program's variables



Evaluation points Examples

- Print the value of a variable to the command line

```
printf("The value of result is %d\n", result);
```

- Skip some code

```
goto 63;
```

- Stop a loop after a certain number of iterations

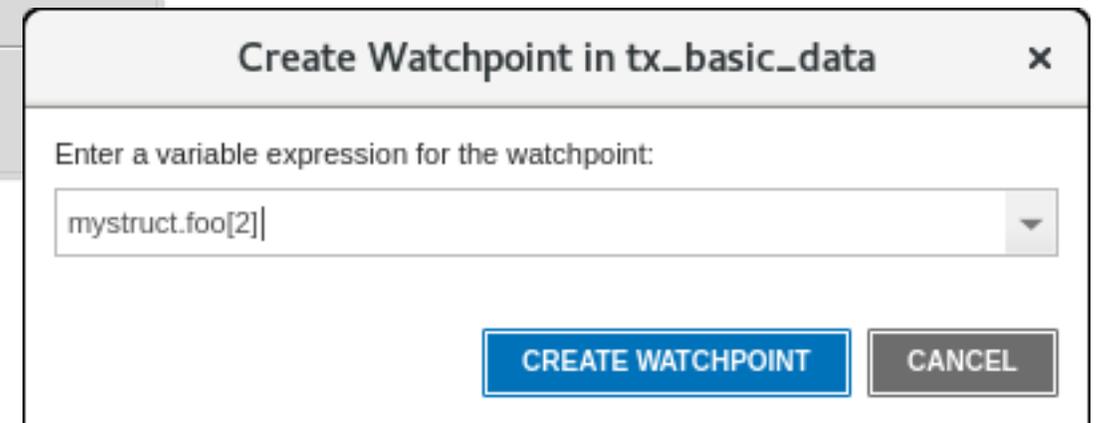
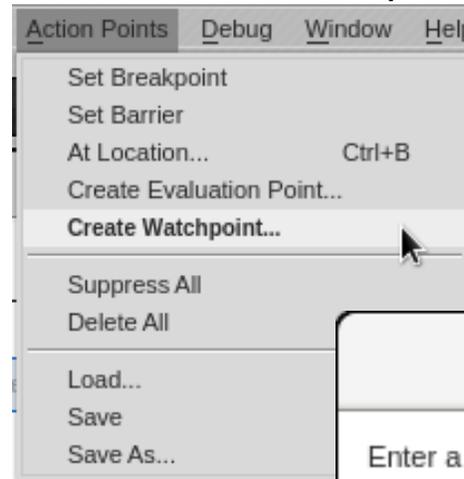
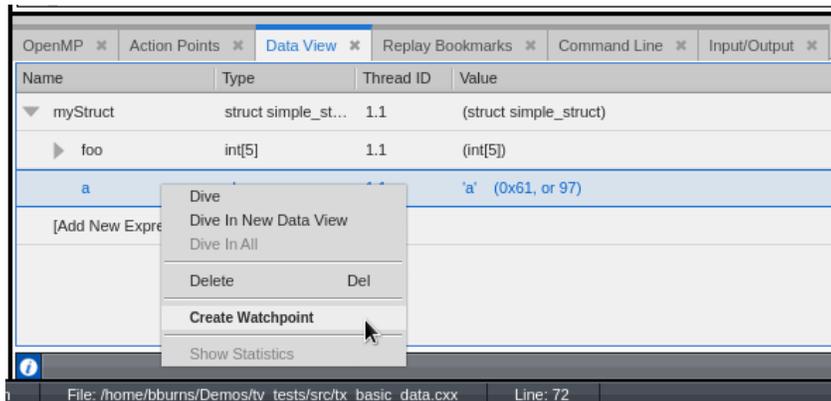
```
if ( (i % 100) == 0) {  
    printf("The value of i is %d\n", i);  
    $stop;  
}
```

See “Using Built-in Statements” in Appendix A of the User Guide for more information on “\$” expressions:

<https://help.totalview.io/current/HTML/index.html#page/TotalView/BuiltInStatments.html#ww1894979>

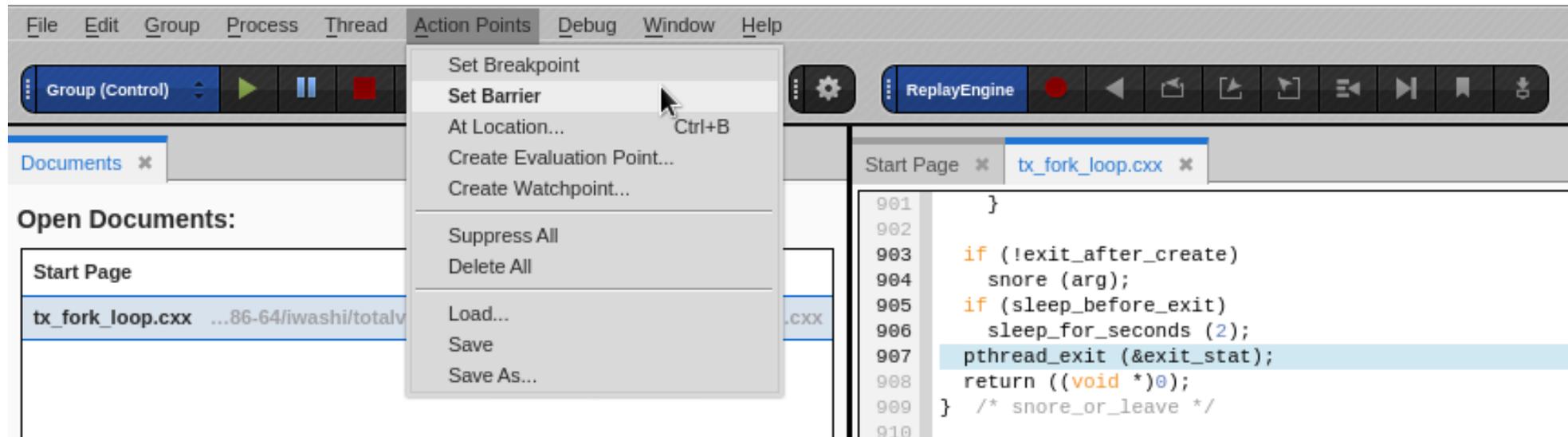
Watchpoints

- Watchpoints are set on a specific memory location
- Execution is stopped when the value stored in that memory location changes
- A breakpoint stops *before* an instruction executes. A watchpoint stops *after* an instruction executes

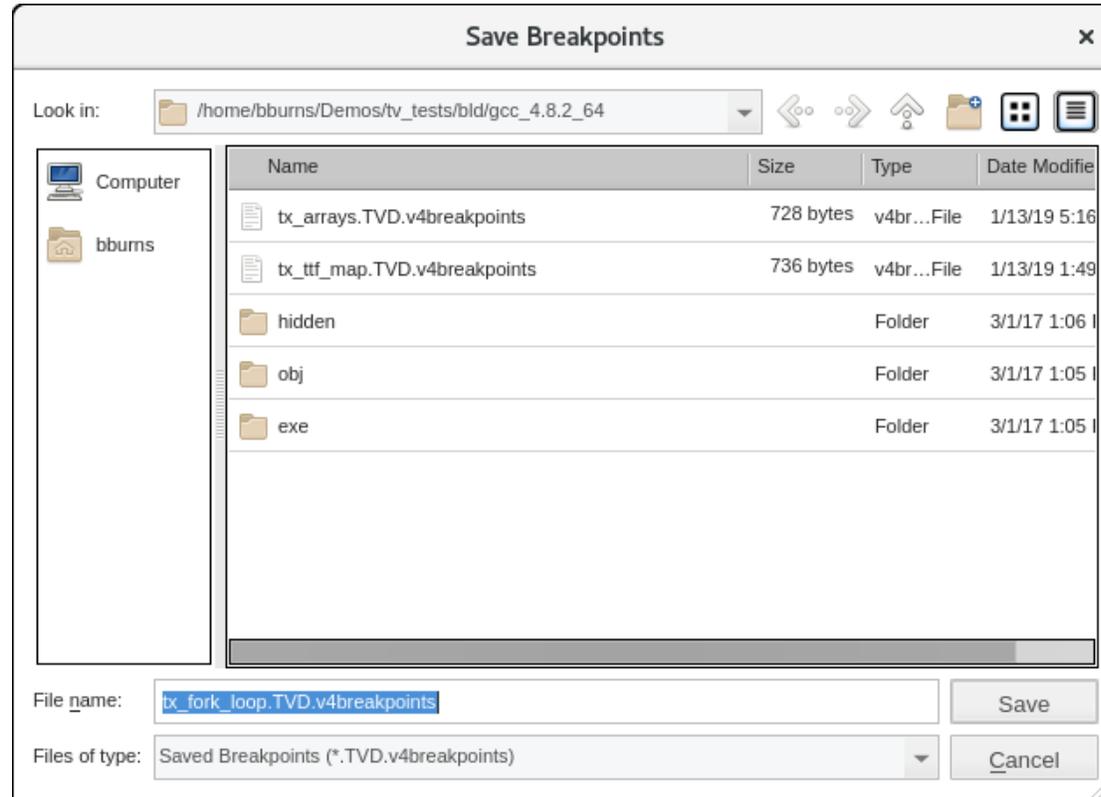


Barrier Breakpoints

- Used to synchronize a group of threads or processes defined in the action point
- Threads or processes are held at barrierpoint until all threads or processes in the group arrive
- When all threads or processes arrive the barrier is satisfied and the threads or processes are released



Saving Breakpoints



From the Action Points menu select Save or Save As to save breakpoints
Turn on option to save action points on exit

Examining and Editing Data

Call Stack and Local Variables

The screenshot displays two panels from the TotalView debugger. The left panel, titled 'Call Stack', shows a list of function calls in a stack. The right panel, titled 'Local Variables', shows the variables for the selected frame.

Name	Type	Value
Arguments		
b	int	0x00000012 (18)
Block at Line 47		
c	int	0x00000014 (20)
i	int	0x00000000 (0)
v	int[20]	(int[20])
[0]	int	0x00000000 (0)
[1]	int	0x00000000 (0)
[2]	int	0x00000000 (0)
[3]	int	0x00000000 (0)
[4]	int	0x00000000 (0)
[5]	int	0x00000000 (0)
[6]	int	0x00000000 (0)
[7]	int	0x00000000 (0)
[8]	int	0x00000000 (0)

Call Stack View

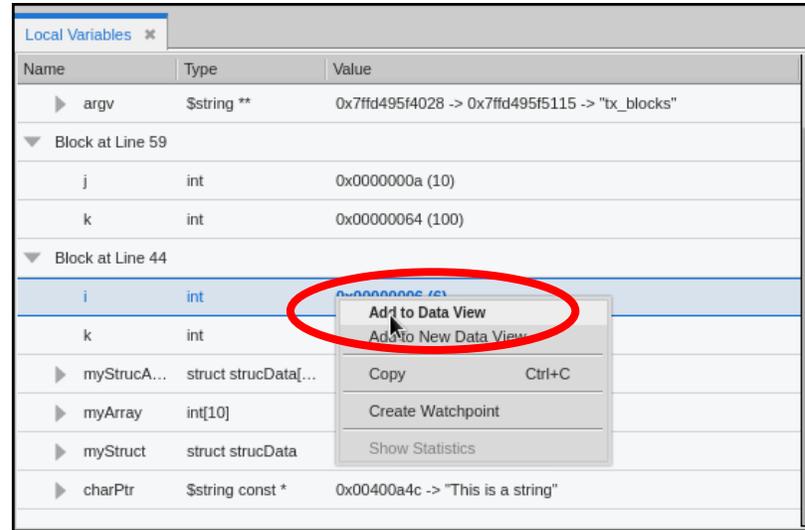
- Lists the set of call frames as the program calls from one function or method to another
- Filter button used to turn on or off filtering of frames.

Local Variables View

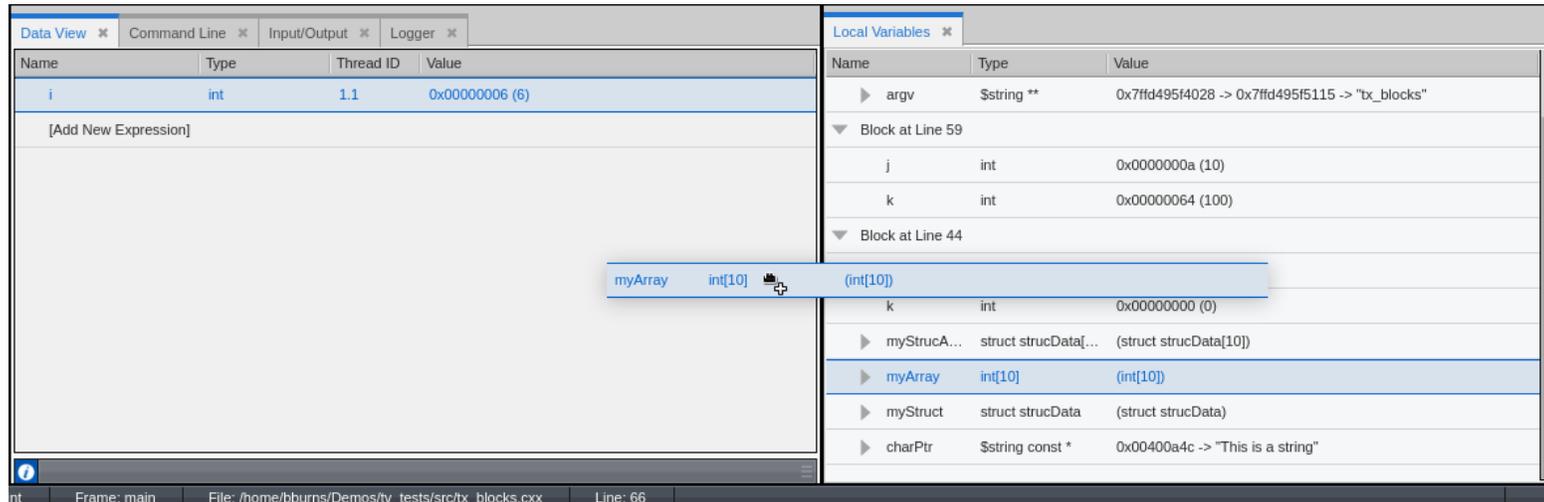
- Displays local variables relative to the current thread of interest and the selected stack frame
- Organized by arguments and blocks
- To edit values, add variable to the Data View

The Data View Panel

- Data View allows deeper exploration of data structures
- Edit data values
- Cast to new data types
- Add data to the Data View using the context menu or by dragging and dropping



Context menu

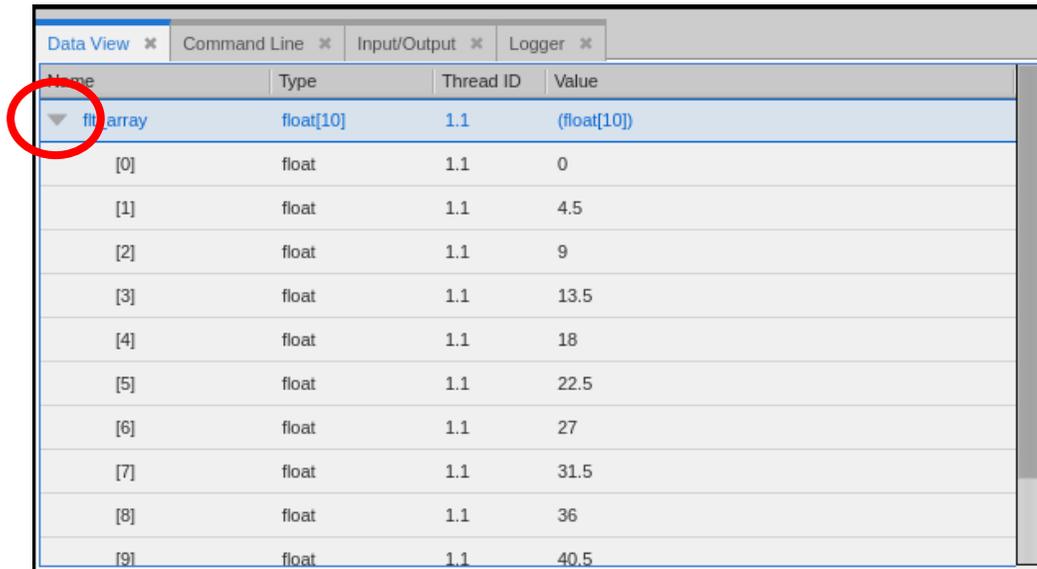


Drag and drop

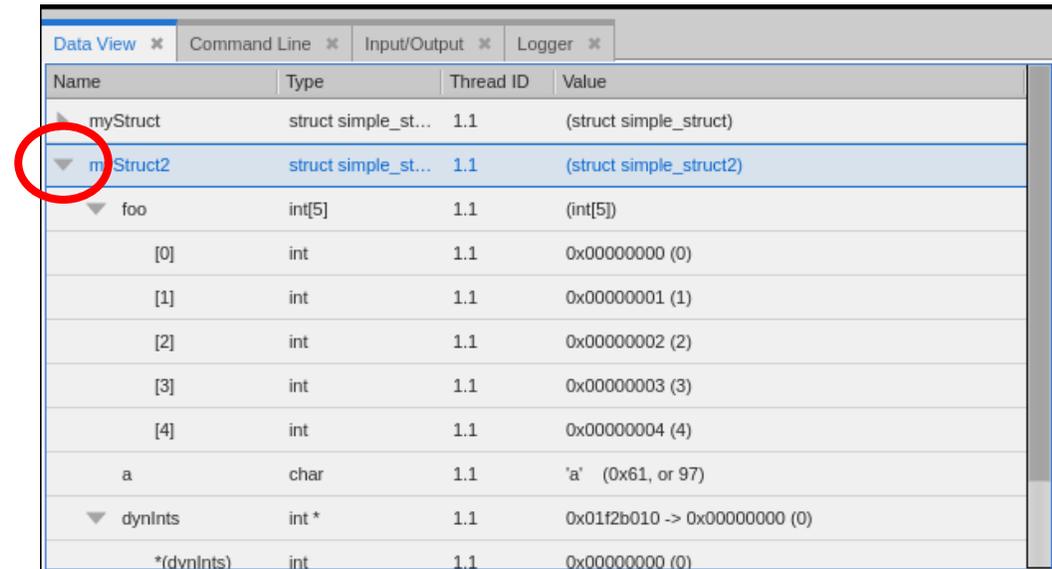
The Data View Panel – Expanding Arrays and Structures

Select the right arrow to display the substructures in a complex variable

Any nested structures are displayed in the data view



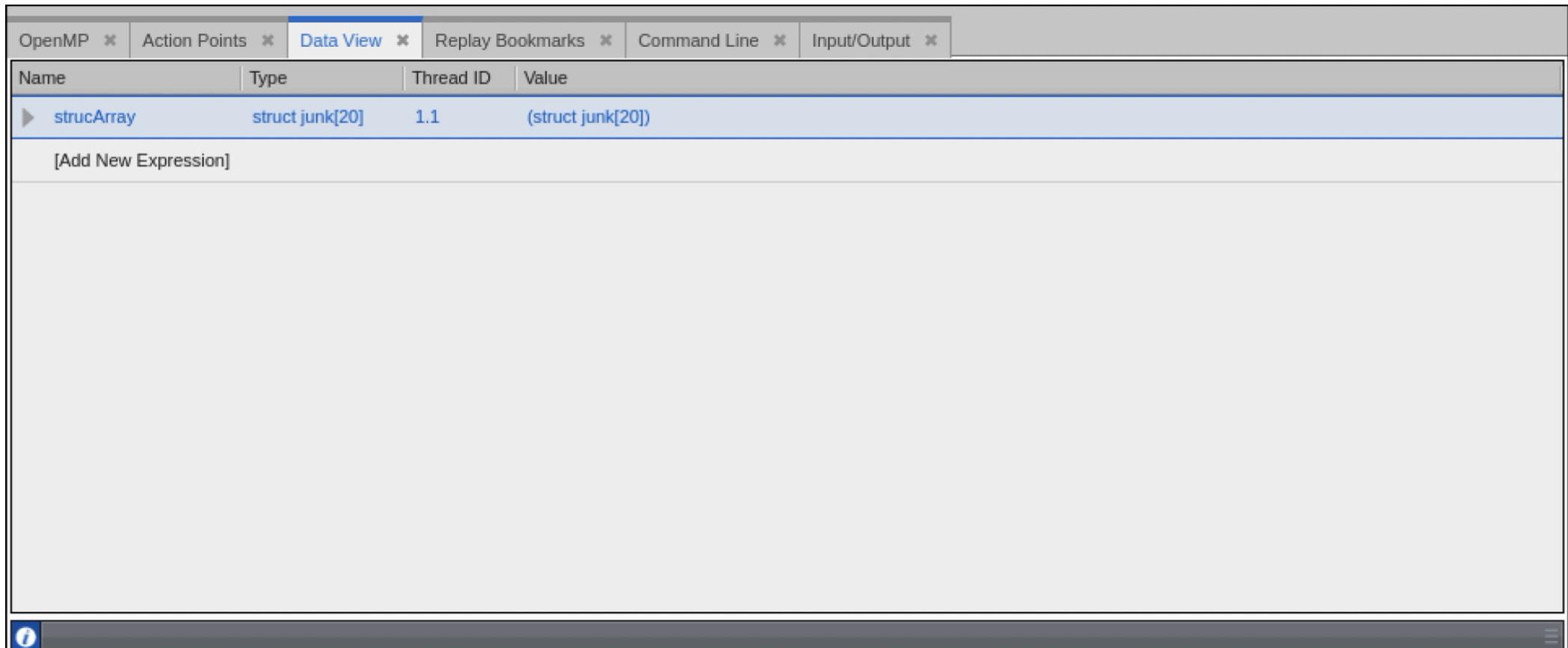
Name	Type	Thread ID	Value
▼ float array	float[10]	1.1	(float[10])
[0]	float	1.1	0
[1]	float	1.1	4.5
[2]	float	1.1	9
[3]	float	1.1	13.5
[4]	float	1.1	18
[5]	float	1.1	22.5
[6]	float	1.1	27
[7]	float	1.1	31.5
[8]	float	1.1	36
[9]	float	1.1	40.5



Name	Type	Thread ID	Value
▶ myStruct	struct simple_st...	1.1	(struct simple_struct)
▼ myStruct2	struct simple_st...	1.1	(struct simple_struct2)
▼ foo	int[5]	1.1	(int[5])
[0]	int	1.1	0x00000000 (0)
[1]	int	1.1	0x00000001 (1)
[2]	int	1.1	0x00000002 (2)
[3]	int	1.1	0x00000003 (3)
[4]	int	1.1	0x00000004 (4)
a	char	1.1	'a' (0x61, or 97)
▼ dynInts	int *	1.1	0x01f2b010 -> 0x00000000 (0)
*dynInts	int	1.1	0x00000000 (0)

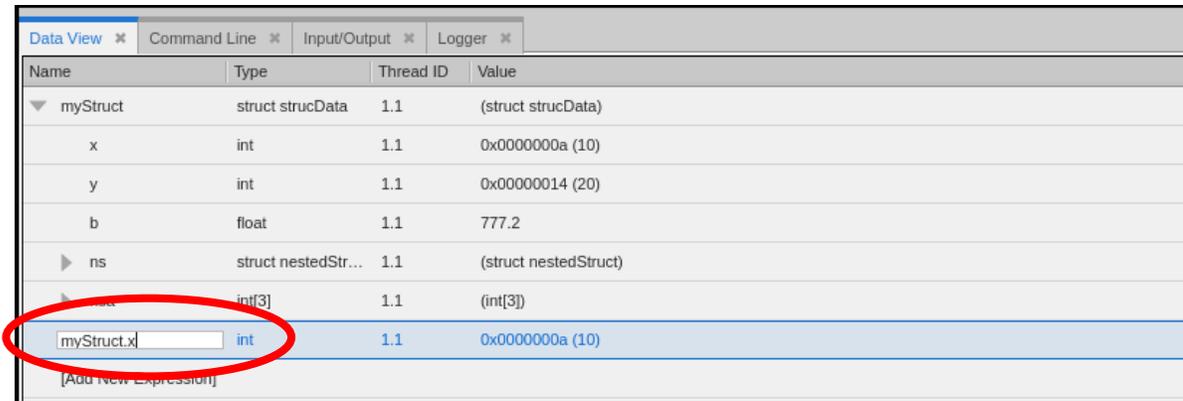
The Data View – Dive in All

- Dive in All
 - Use Dive in All to easily see each member of a data structure from an array of structures



The Data View Panel – Entering Expressions

Enter a new expression in the Data View panel to view that data



Type the expression in the [Add New expression] field



A new expression is added



Increment a variable

The Data View Panel - Casting

Casting to another type



A screenshot of the TotalView interface showing the array view for variable 'j'. The variable 'j' is circled in blue, and a blue arrow points down to the array view. The array view shows three elements: [0], [1], and [2].

	int[3]	(int[3])
[0]	int	0x0000000a (10)
[1]	int	0x00000064 (100)
[2]	int	0x00000010 (16)

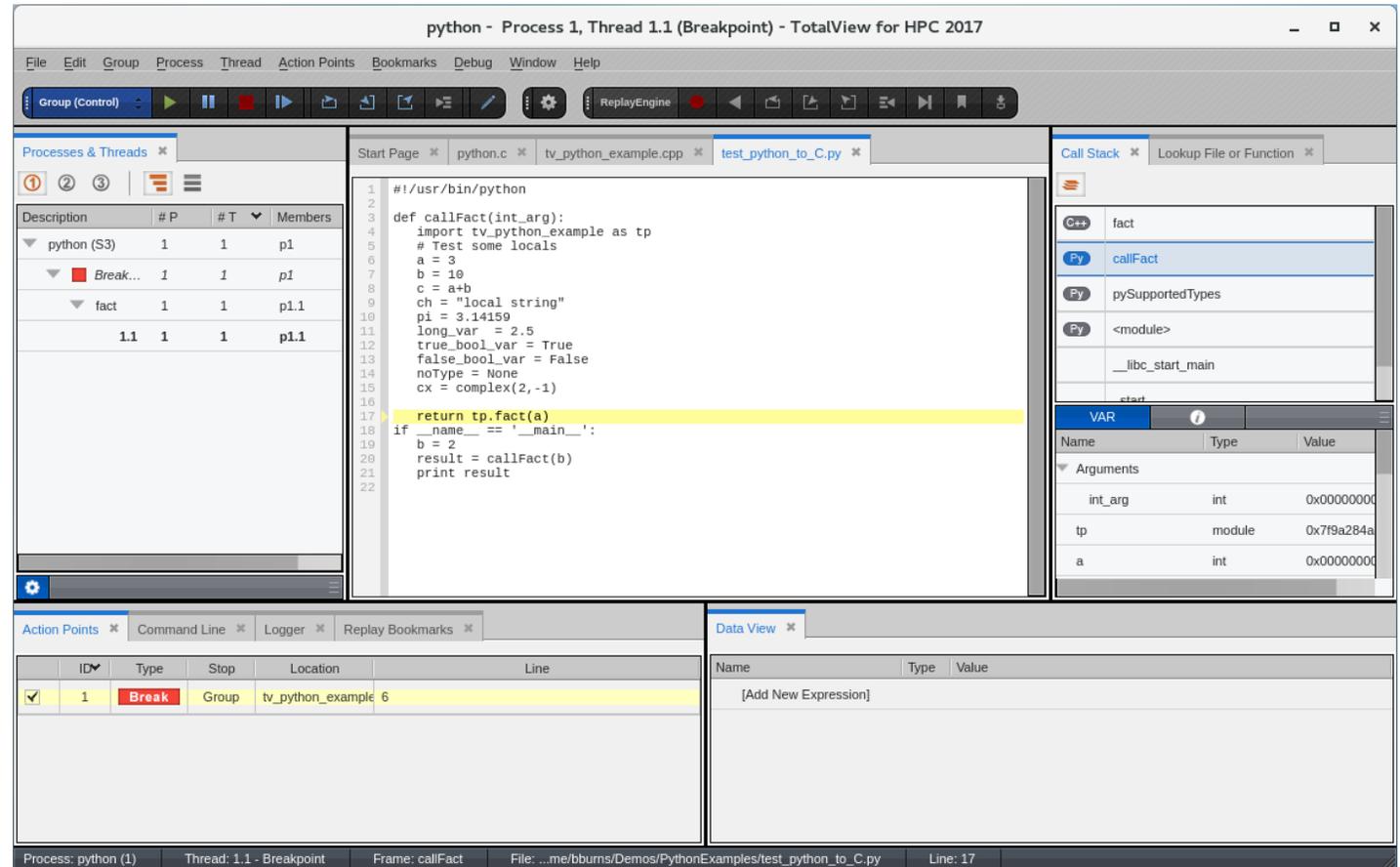
Cast a variable into an array by adding the array specifier

TotalView displays the array

Extending Debugging Capabilities: How to Debug (AI) Mixed Python/C++ Code

Mixed Language Python Debugging

- Debugging one language is difficult enough.
- Understanding the flow of execution across language barriers is hard.
- Examining and comparing data in both languages is challenging.
- What TotalView provides:
 - Easy python debugging session setup.
 - Fully integrated Python and C/C++ call stack.
 - "Glue" layers between the languages removed.
 - Easily examine and compare variables in Python and C++.
 - Modest system requirements.
 - Utilize reverse debugging and memory debugging.
- What TotalView does not provide (yet):
 - Setting breakpoints and stepping within Python code.



Python debugging with TotalView (demo)

```
#!/usr/bin/python

def callFact():
    import tv_python_example as tp
    a = 3
    b = 10
    c = a+b
    ch = "local string"
    .....
    return tp.fact(a)
if __name__ == '__main__':
    b = 2
    result = callFact()
    print result
```

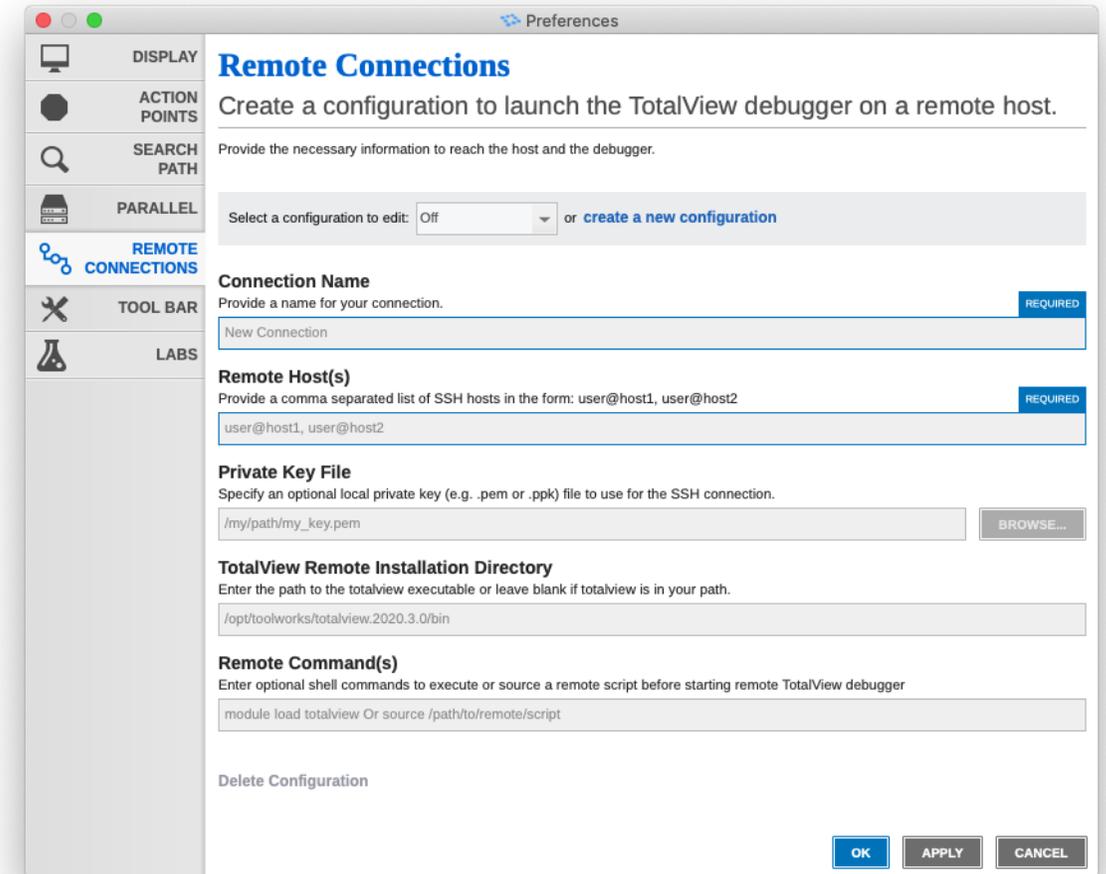


```
Terminal  
ubuntu:~/demo_2019/PythonExamples> /usr/toolworks/totalview.2019.0.4/bin/totalvi  
ew -args python2.7-dbg test_python_types.py
```

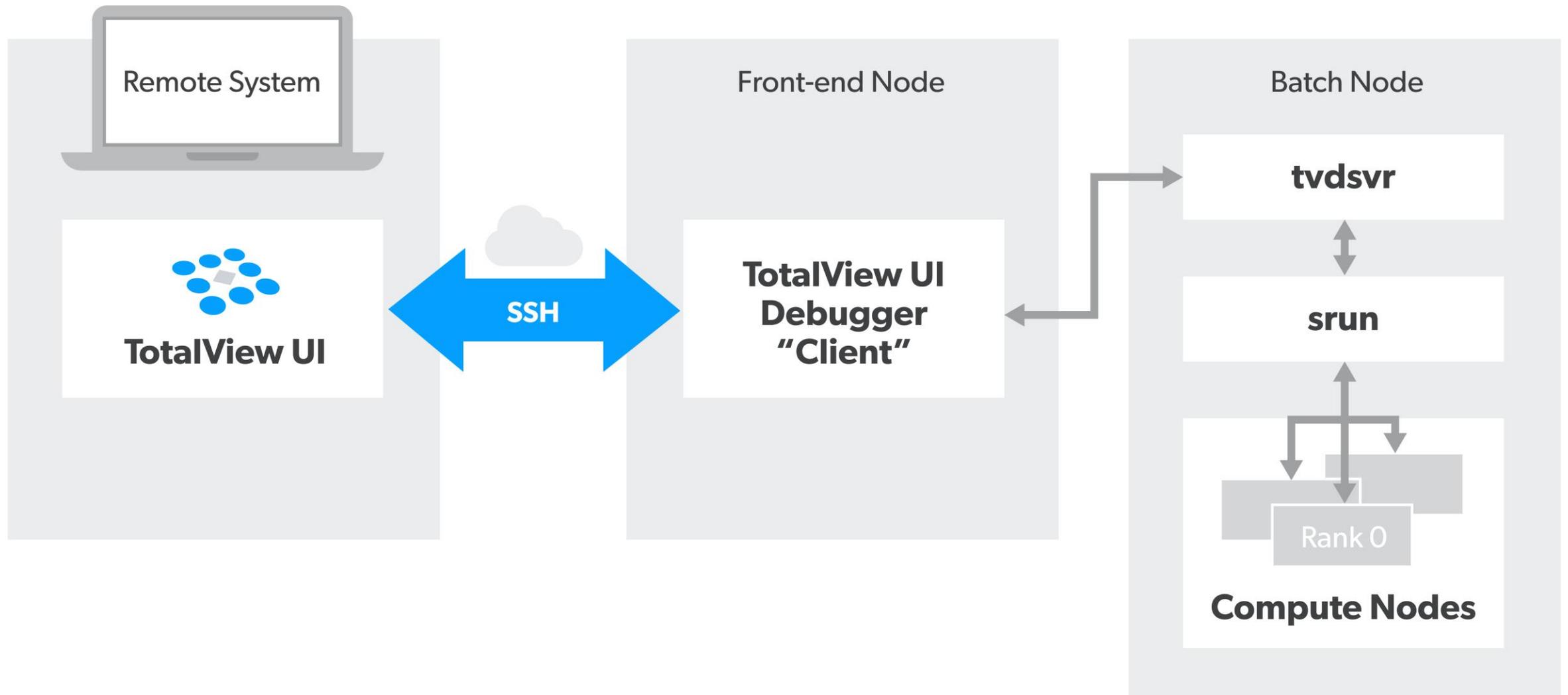


TotalView Remote UI

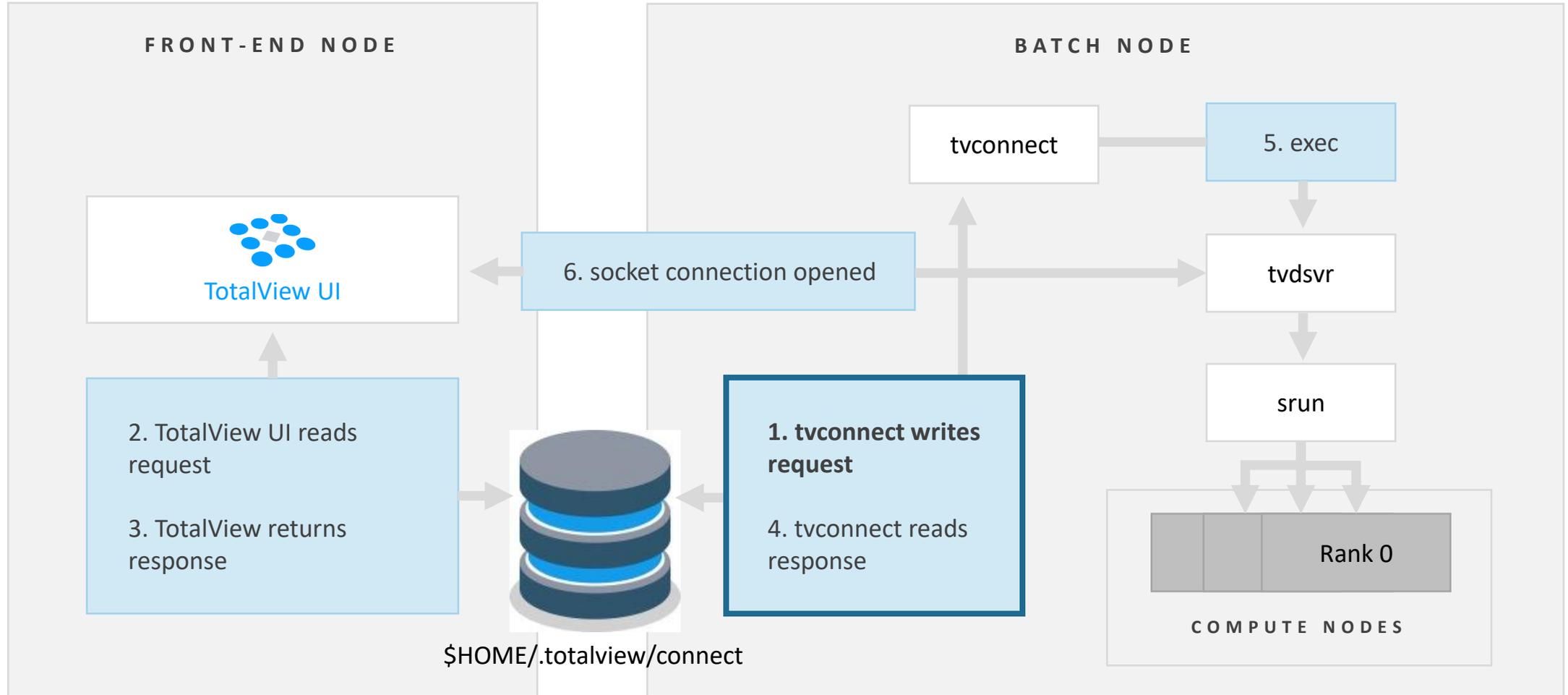
- Combine the convenience of establishing a remote connection to a cluster and the ability to run the TotalView GUI locally.
- Front-end GUI architecture does not need to match back-end target architecture (macOS front-end -> Linux back-end)
- Secure communications
- Convenient saved sessions
- Once connected, debug as normal with access to all TotalView features
- Front-end GUI currently supports macOS and Linux x86/x86-64. Windows client is coming.



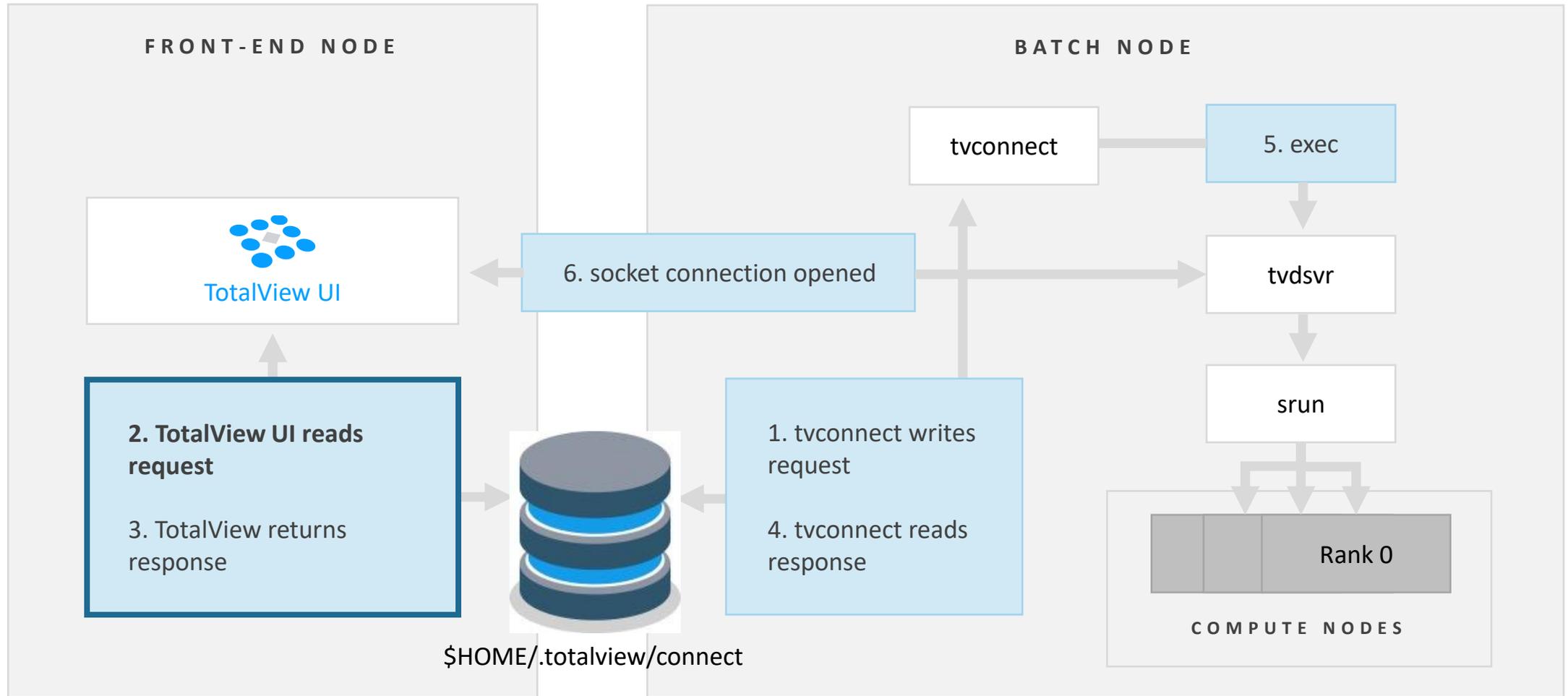
Remote UI Architecture



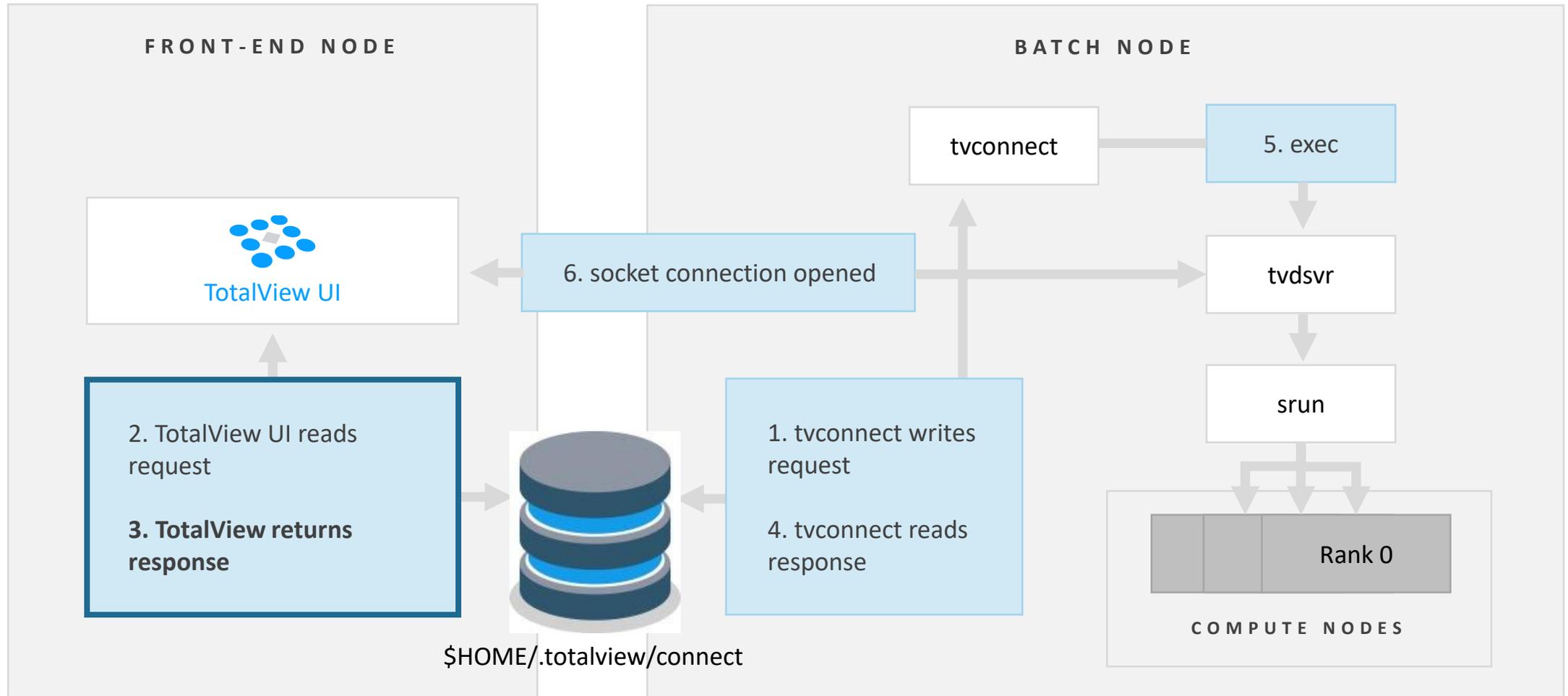
Reverse Connection Flow



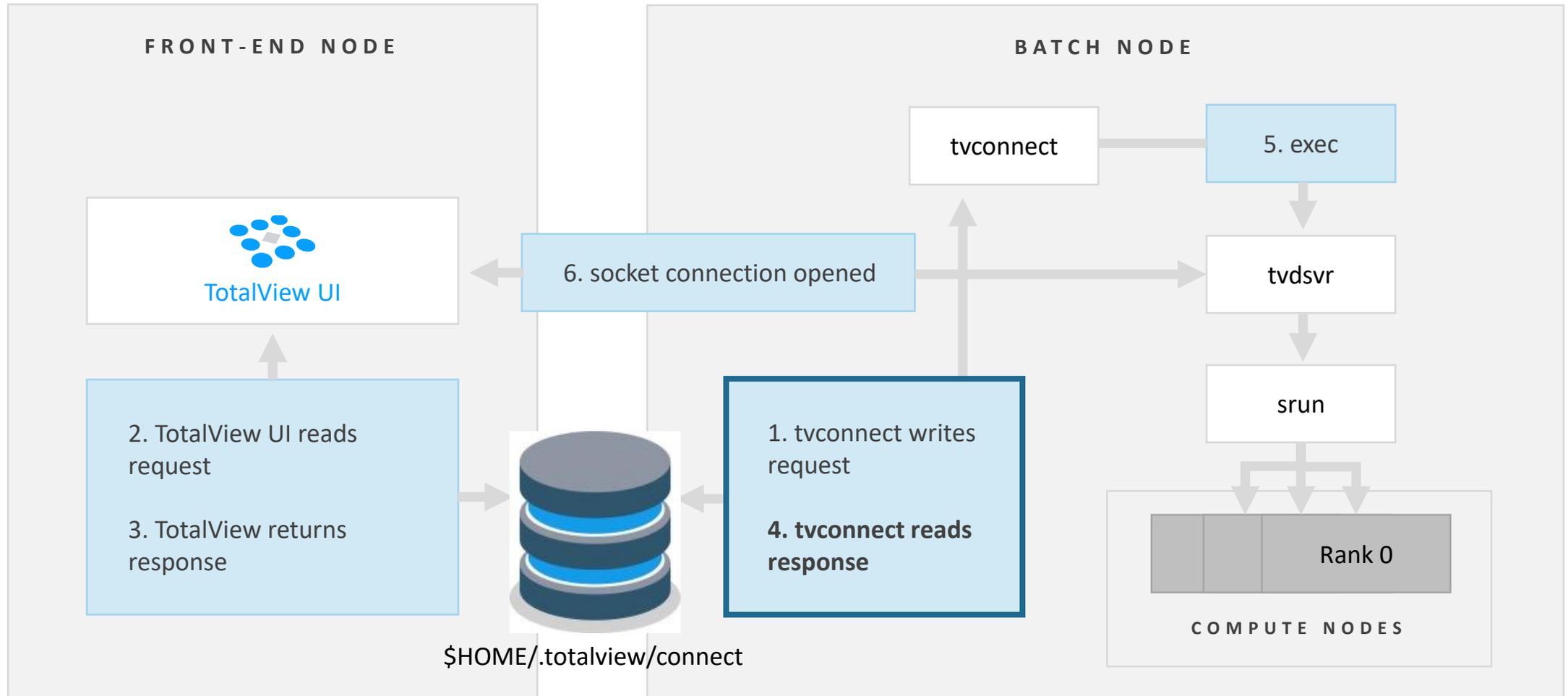
Reverse Connection Flow



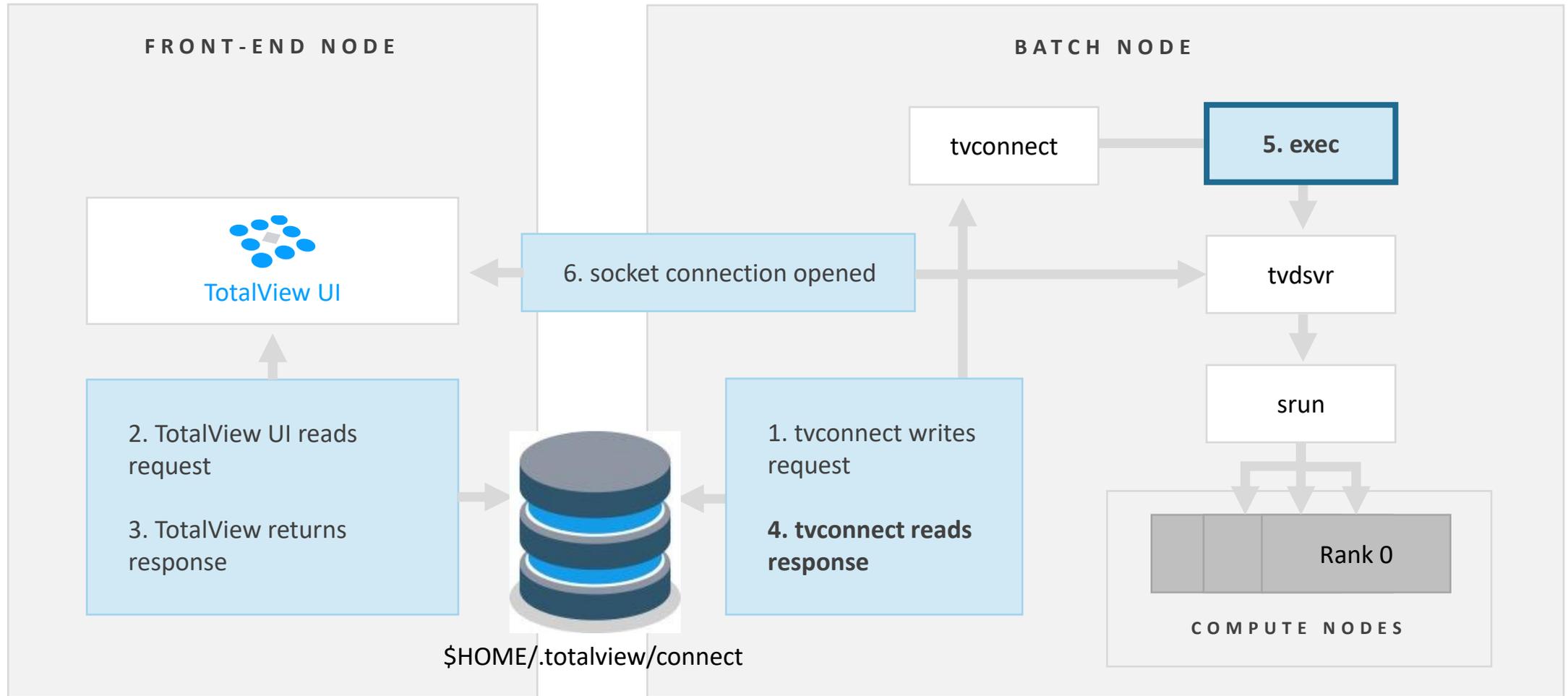
Reverse Connection Flow



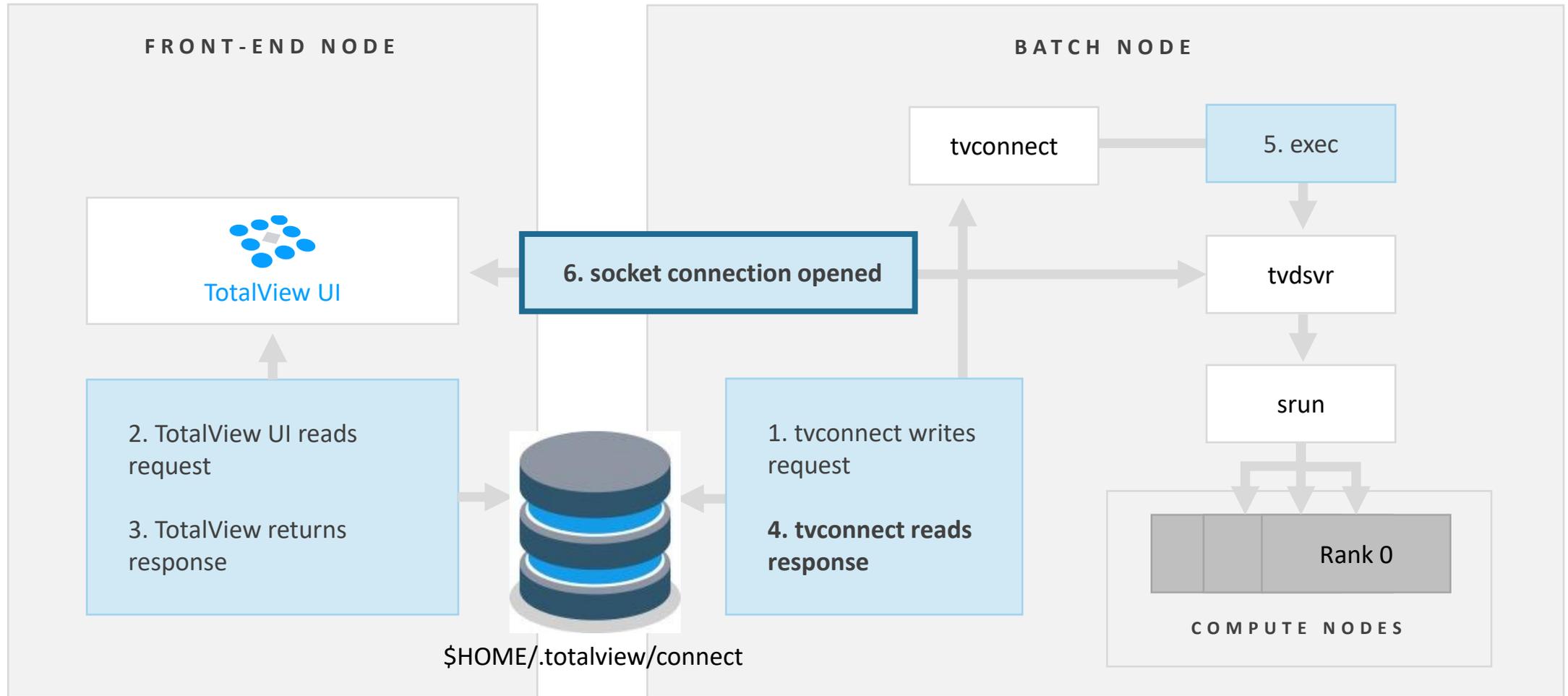
Reverse Connection Flow



Reverse Connection Flow



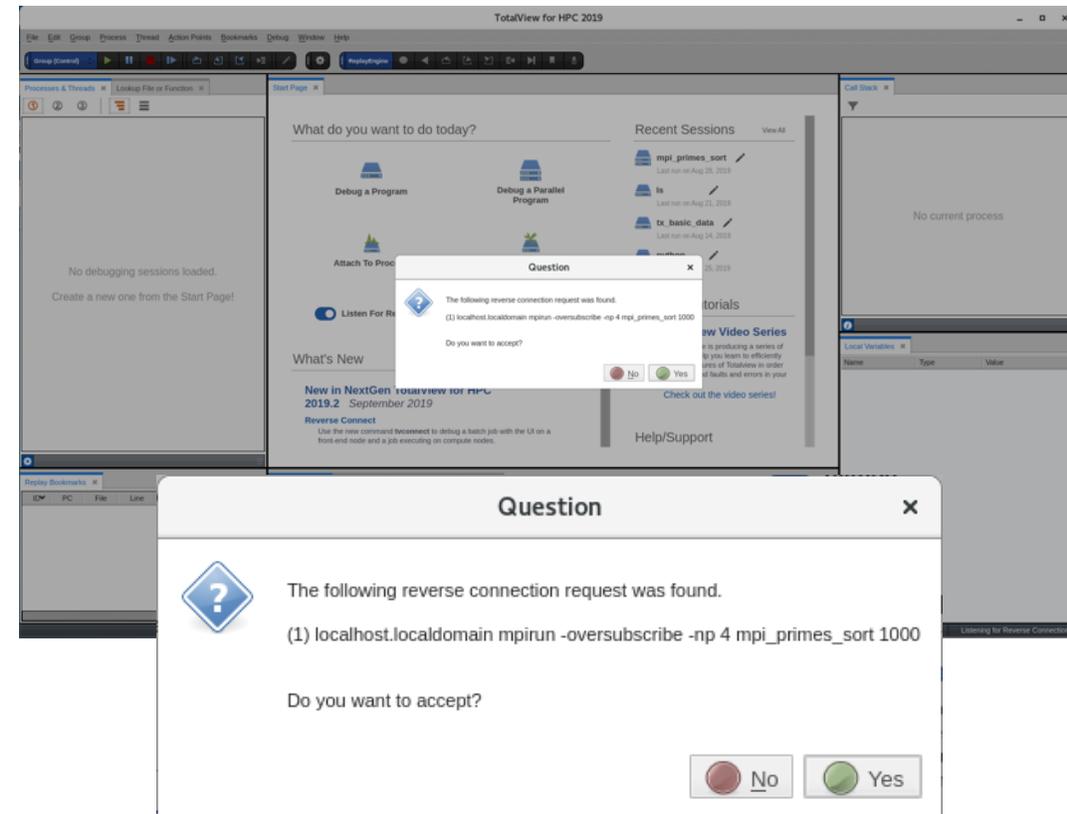
Reverse Connection Flow



Batch Script Submission with Reverse Connect

- Start a debugging session using TotalView Reverse Connect.
- Reverse Connect enables the debugger to be submitted to a cluster and connected to the GUI once run.
- Enables running TotalView UI on the front-end node and remotely debug jobs executing on the compute nodes.
- Very easy to utilize, simply prefix job launch or application start with “tvconnect” command.

```
#!/bin/bash
#SBATCH -J hybrid_fib
...
#SBATCH -n 2
#SBATCH -c 4
#SBATCH --mem-per-cpu=4000
export OMP_NUM_THREADS=4
tvconnect srun -n 2 --cpus-per-task=4 --mpi=pmix ./hybrid_fib
```



Memory Leaks, Heap Status, and Identifying Dangling Pointers

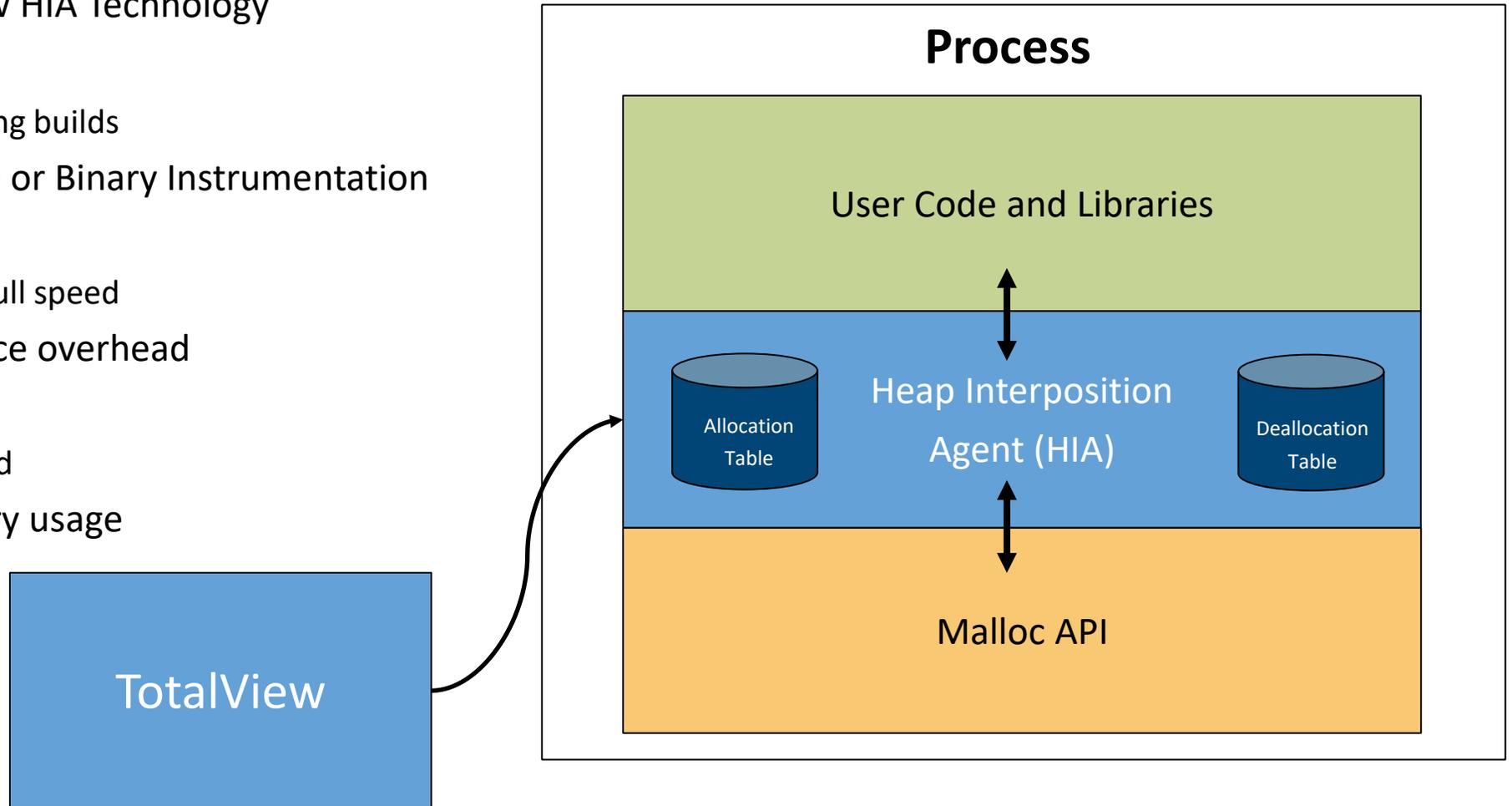
What is a Memory Bug?

- A Memory Bug is a mistake in the management of heap memory
 - Leaking: Failure to free memory
 - Dangling references: Failure to clear pointers
 - Failure to check for error conditions
 - Memory Corruption
 - Writing to memory not allocated
 - Overrunning array bounds



TotalView Heap Interposition Agent (HIA) Technology

- Advantages of TotalView HIA Technology
 - Use it with your existing builds
 - No Source Code or Binary Instrumentation
 - Programs run nearly full speed
 - Low performance overhead
 - Low memory overhead
 - Efficient memory usage



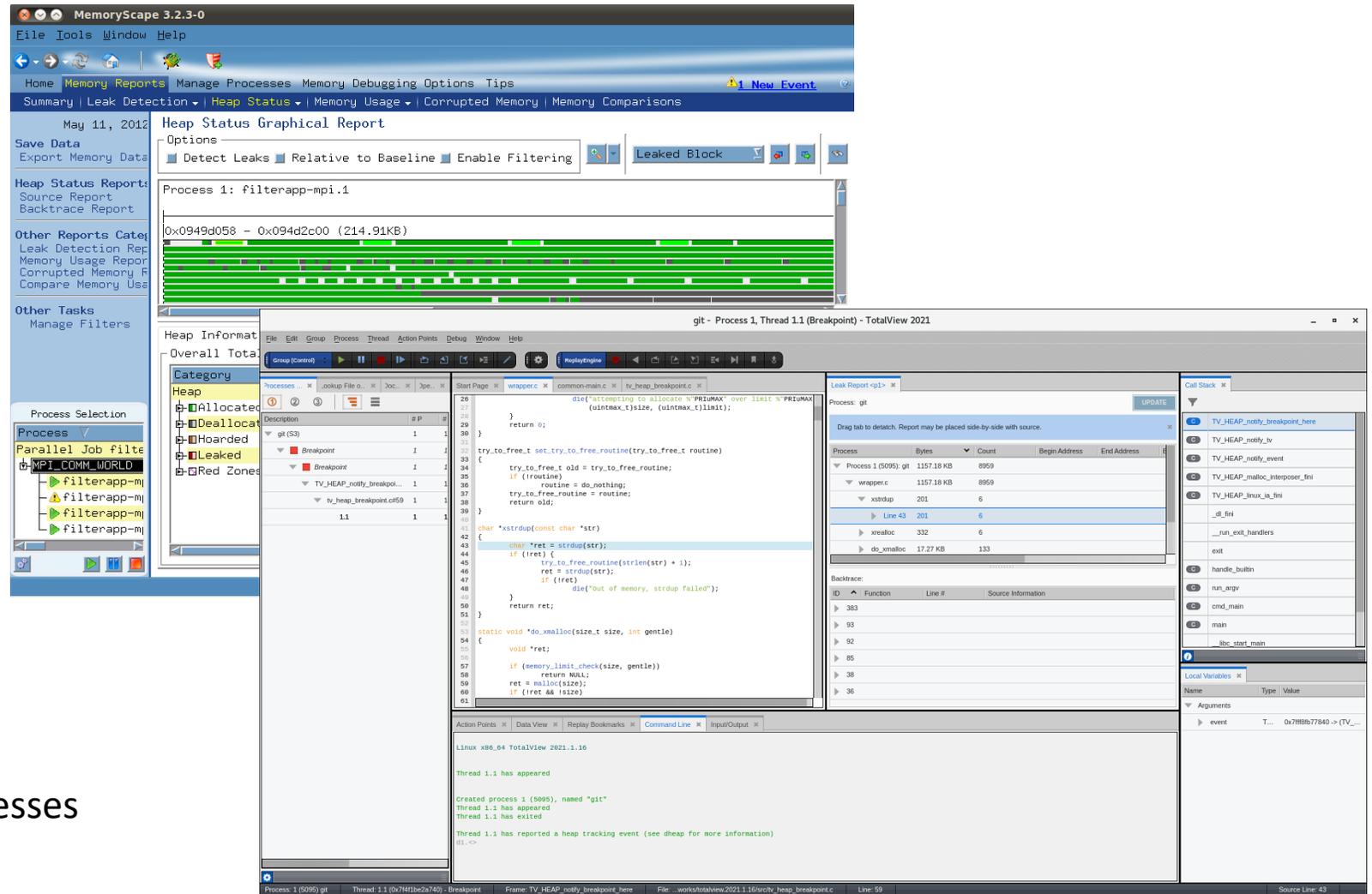
Memory Debugging Features – MemoryScope / TotalView

TotalView New UI Features

- Leak detection
- Heap Status
- Dangling pointer detection

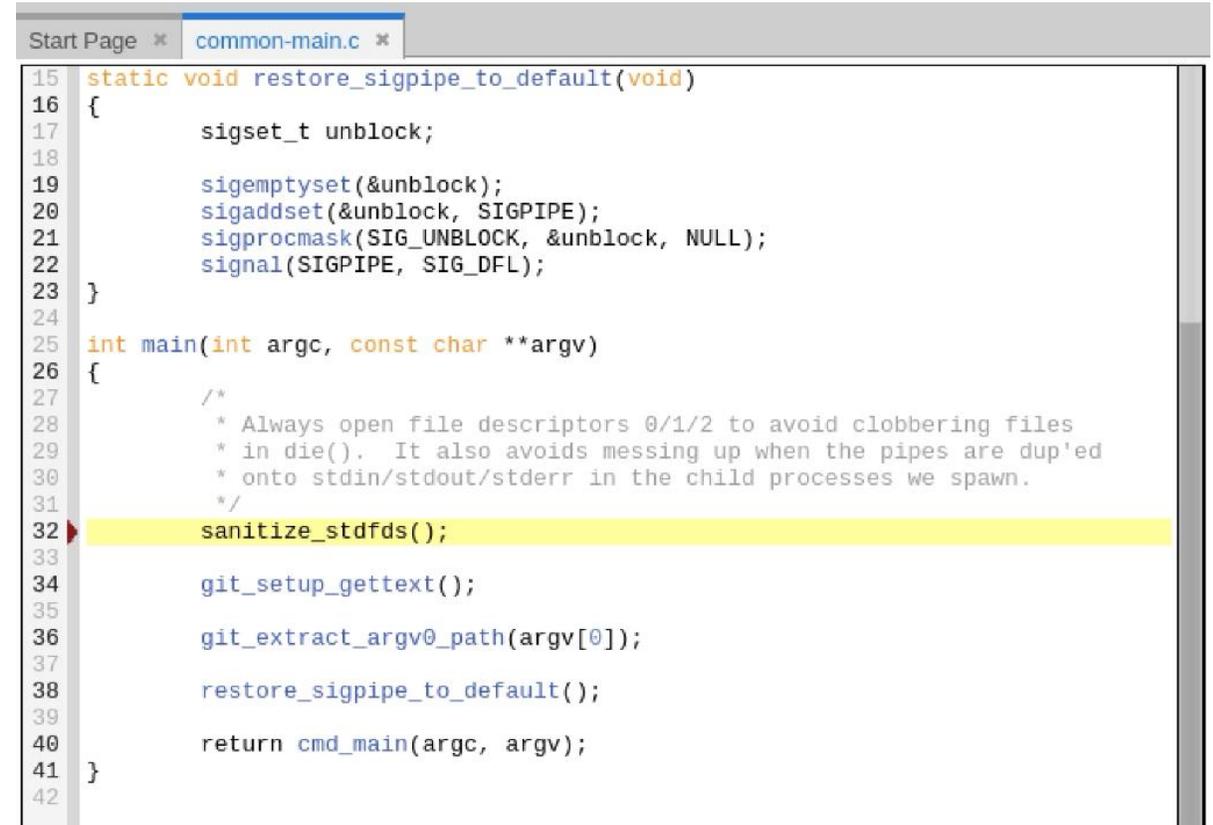
Coming Features

- Memory Error Events
- Memory Corruption Detection
- Memory Block Painting
- Memory Hoarding
- Memory Comparisons between processes



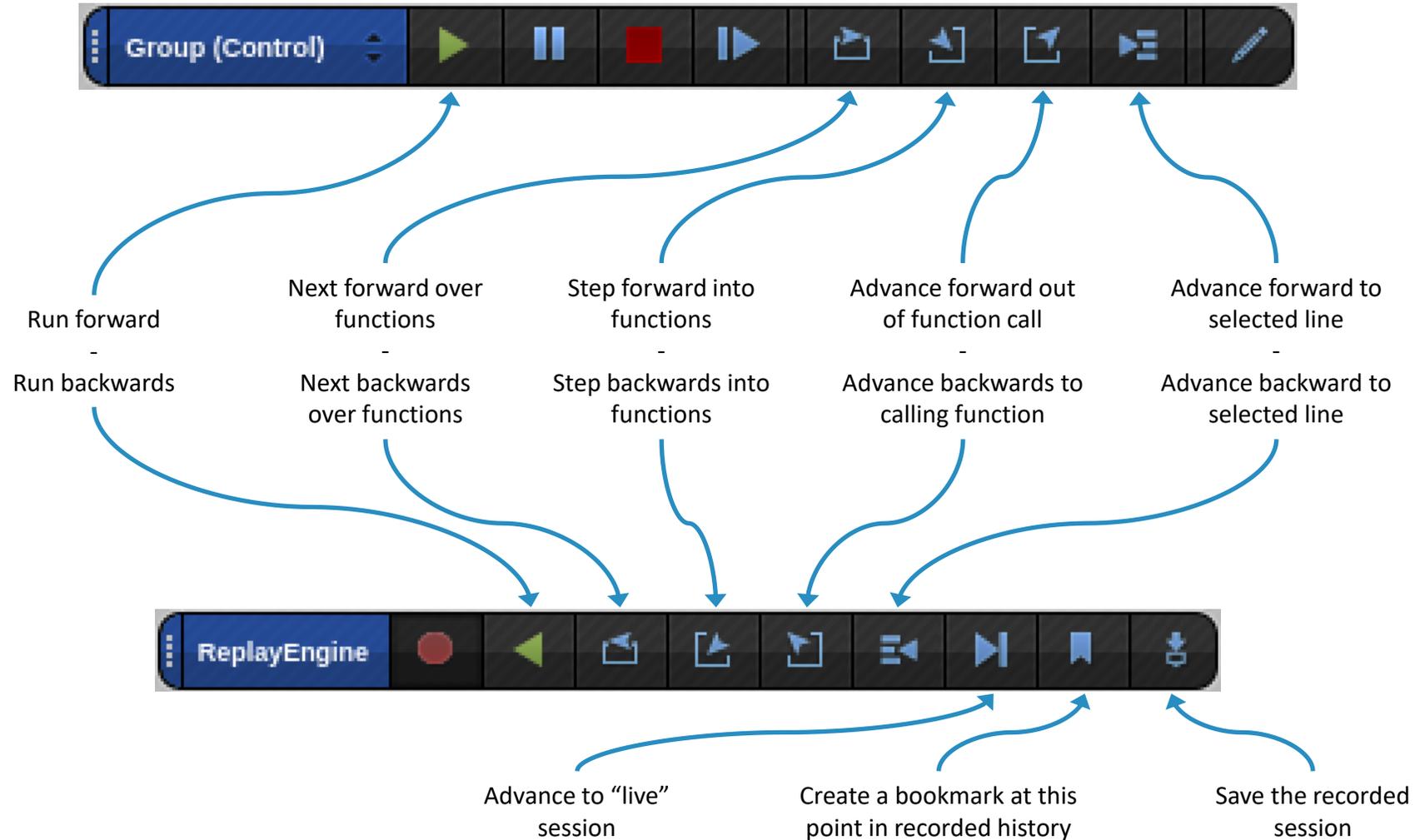
Reverse Debugging with TotalView

- Reverse debugging provides the ability for developers to go back in execution history
- Activated either before program starts running or at some point after execution begins.
- Capturing and deterministically replay execution.
- Enables stepping backwards and forward by function, line or instruction.
- Run backwards to breakpoints.
- Run backwards and stop when a variable changes value.
- Saving recording files for later analysis or collaboration.
 - For remote connection use CLI: `dhistory -save <name>`



```
Start Page * common-main.c *
15 static void restore_sigpipe_to_default(void)
16 {
17     sigset_t unblock;
18
19     sigemptyset(&unblock);
20     sigaddset(&unblock, SIGPIPE);
21     sigprocmask(SIG_UNBLOCK, &unblock, NULL);
22     signal(SIGPIPE, SIG_DFL);
23 }
24
25 int main(int argc, const char **argv)
26 {
27     /*
28      * Always open file descriptors 0/1/2 to avoid clobbering files
29      * in die(). It also avoids messing up when the pipes are dup'ed
30      * onto stdin/stdout/stderr in the child processes we spawn.
31      */
32     sanitize_std fds();
33
34     git_setup_gettext();
35
36     git_extract_argv0_path(argv[0]);
37
38     restore_sigpipe_to_default();
39
40     return cmd_main(argc, argv);
41 }
42
```

Reverse Debugging Controls



Debugging CUDA Applications with TotalView

TotalView for the NVIDIA[®] GPU Accelerator

- NVIDIA Tesla, Fermi, Kepler, Pascal, Volta, Turing, Ampere
- NVIDIA CUDA 9.2, 10 and 11
 - With support for Unified Memory
- Debugging 64-bit CUDA programs
- Features and capabilities include
 - Support for dynamic parallelism
 - Support for MPI based clusters and multi-card configurations
 - Flexible Display and Navigation on the CUDA device
 - Physical (device, SM, Warp, Lane)
 - Logical (Grid, Block) tuples
 - CUDA device window reveals what is running where
 - Support for types and separate memory address spaces
 - Leverages CUDA memcheck

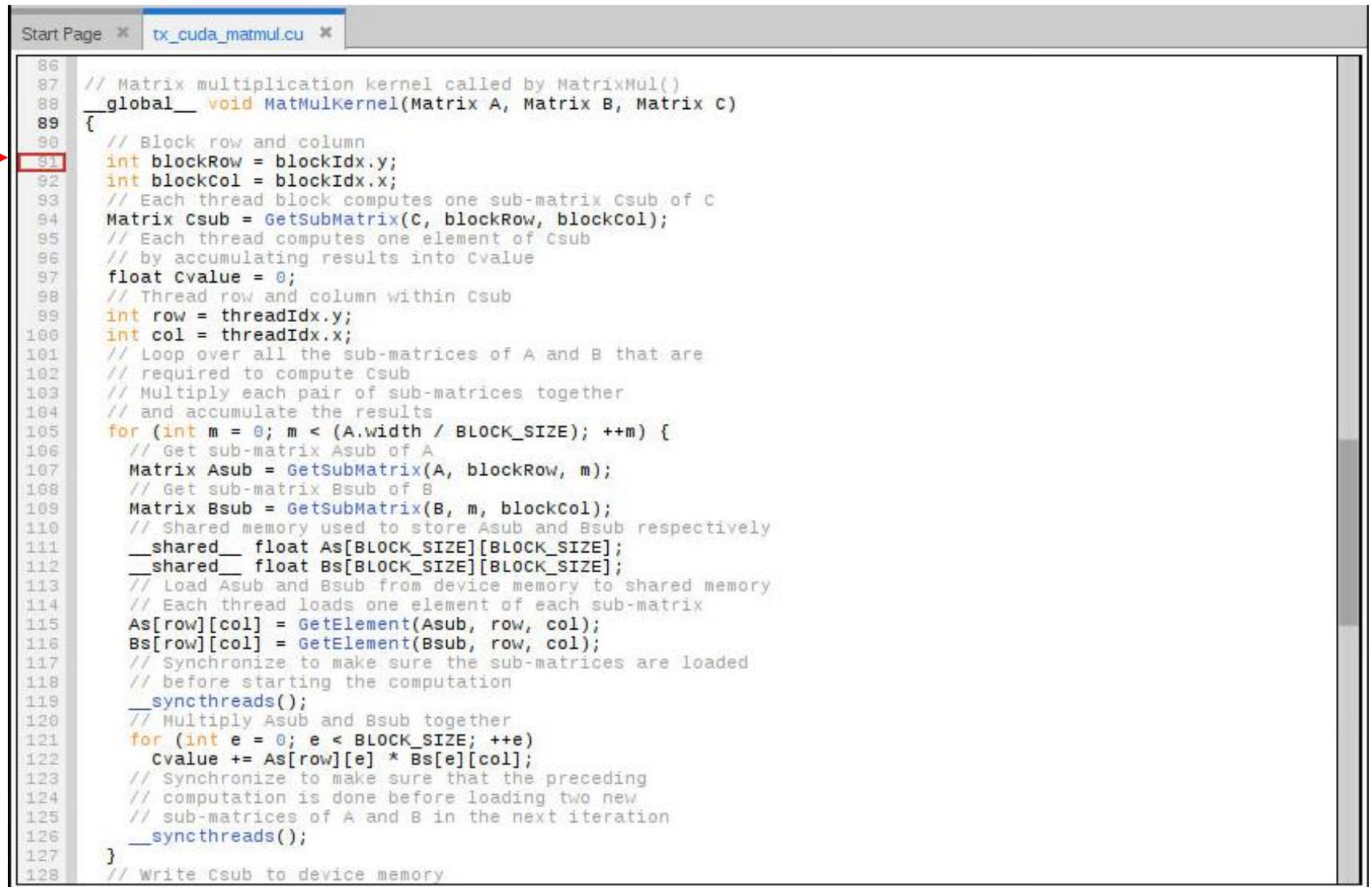


Source View Opened on CUDA host code

```
Start Page * tx_cuda_matmul.cu *
139 Matrix A;
140 A.width = width_;
141 A.height = height_;
142 A.stride = width_;
143 A.elements = (float*) malloc(sizeof(*A.elements) * width_ * height_);
144 for (int row = 0; row < height_; row++)
145     for (int col = 0; col < width_; col++)
146         A.elements[row * width_ + col] = row * 10.0 + col;
147 return A;
148 }
149
150 static void
151 print_Matrix (Matrix A, const char *name)
152 {
153     printf("%s:\n", name);
154     for (int row = 0; row < A.height; row++)
155         for (int col = 0; col < A.width; col++)
156             printf ("%5d[%5d] %f\n", row, col, A.elements[row * A.stride + col]);
157 }
158
159 // Multiply an m*n matrix with an n*p matrix results in an m*p matrix.
160 // Usage: tx_cuda_matmul [ m [ n [ p ] ] ]
161 // m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
162 int main(int argc, char **argv)
163 {
164     cudaSetDevice(0);
165     const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
166     const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
167     const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
168     Matrix A = cons_Matrix(m, n);
169     Matrix B = cons_Matrix(n, p);
170     Matrix C = cons_Matrix(m, p);
171     MatMul(A, B, C);
172     print_Matrix(A, "A");
173     print_Matrix(B, "B");
174     print_Matrix(C, "C");
175     return 0;
176 }
177
178 /*
179  * Update log
180  *
181  * Feb 25 2015 NYP: Removed forceinline , it is making cli too fast
```

Breakpoint Set in CUDA Kernel Code Before Launch

Hollow breakpoint indicates a breakpoint will be set when the code is loaded onto the GPU.



```
86 // Matrix multiplication kernel called by MatrixMul()
87
88 __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
89 {
90     // Block row and column
91     int blockRow = blockIdx.y;
92     int blockCol = blockIdx.x;
93     // Each thread block computes one sub-matrix Csub of C
94     Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
95     // Each thread computes one element of Csub
96     // by accumulating results into Cvalue
97     float Cvalue = 0;
98     // Thread row and column within Csub
99     int row = threadIdx.y;
100    int col = threadIdx.x;
101    // Loop over all the sub-matrices of A and B that are
102    // required to compute Csub
103    // Multiply each pair of sub-matrices together
104    // and accumulate the results
105    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
106        // Get sub-matrix Asub of A
107        Matrix Asub = GetSubMatrix(A, blockRow, m);
108        // Get sub-matrix Bsub of B
109        Matrix Bsub = GetSubMatrix(B, m, blockCol);
110        // Shared memory used to store Asub and Bsub respectively
111        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
112        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
113        // Load Asub and Bsub from device memory to shared memory
114        // Each thread loads one element of each sub-matrix
115        As[row][col] = GetElement(Asub, row, col);
116        Bs[row][col] = GetElement(Bsub, row, col);
117        // Synchronize to make sure the sub-matrices are loaded
118        // before starting the computation
119        __syncthreads();
120        // Multiply Asub and Bsub together
121        for (int e = 0; e < BLOCK_SIZE; ++e)
122            Cvalue += As[row][e] * Bs[e][col];
123        // Synchronize to make sure that the preceding
124        // computation is done before loading two new
125        // sub-matrices of A and B in the next iteration
126        __syncthreads();
127    }
128    // Write Csub to device memory
```

GPU Physical and Logical Toolbars



Logical toolbar displays the Block and Thread coordinates.

Physical toolbar displays the Device number, Streaming Multiprocessor, Warp and Lane.

To view a CUDA host thread, select a thread with a positive thread ID in the Process and Threads view.

To view a CUDA GPU thread, select a thread with a negative thread ID, then use the GPU thread selector on the logical toolbar to focus on a specific GPU thread.

Displaying CUDA Program Elements

Name	Type	Thread ID	Value
▼ A	Matrix @local	1-1	(Matrix @local)
width	int	1-1	0x00000002 (2)
height	int	1-1	0x00000002 (2)
stride	int	1-1	0x00000002 (2)
▼ elements	float @generic *	1-1	0x7f724e800000 -> 0
*(elements)	@generic float	1-1	0
[Add New Expression]			

@local type qualifier indicates that variable A is in local storage.

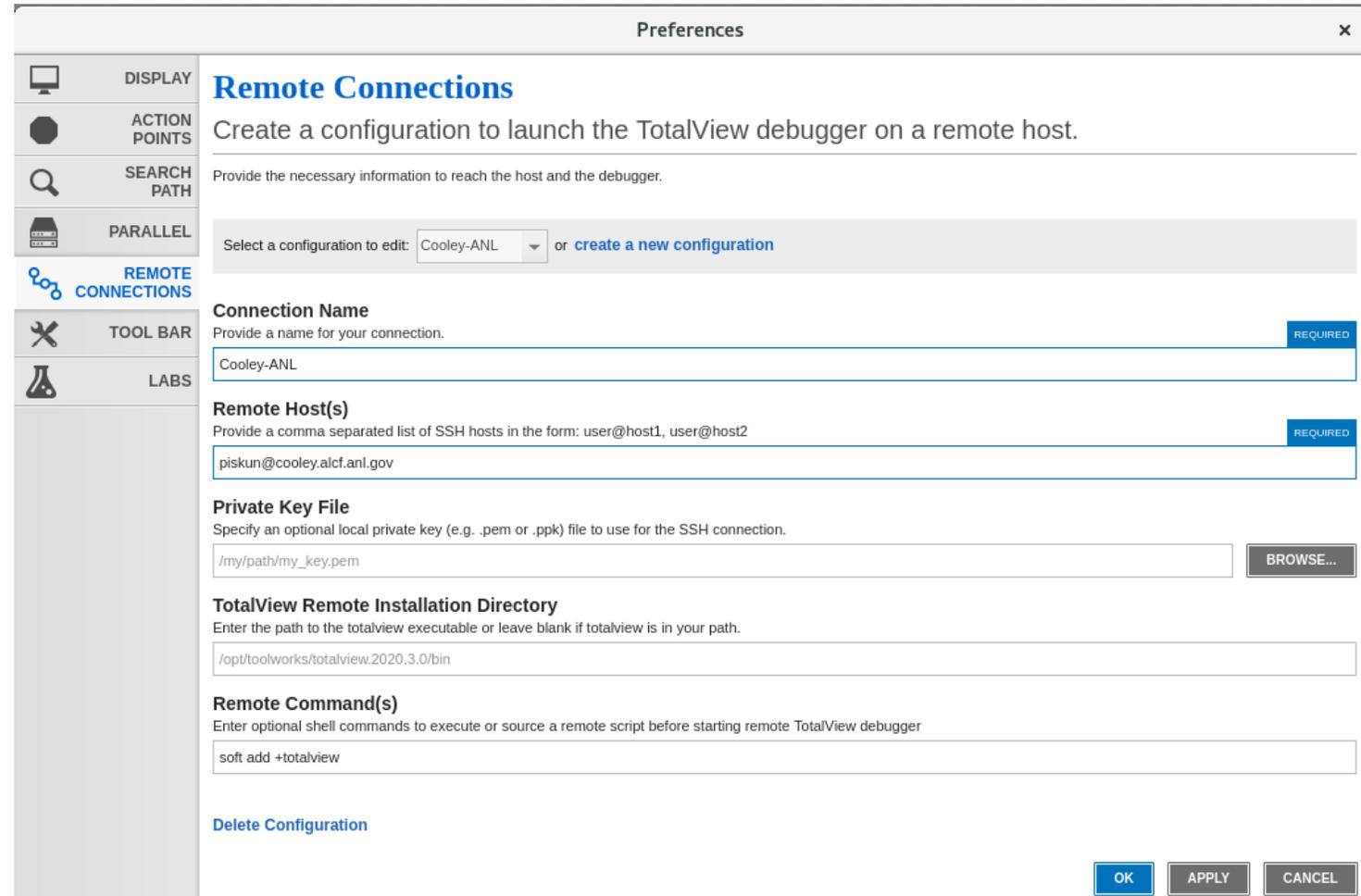
"elements" is a pointer to a float in @generic storage.

- The identifier @local is a TotalView built-in type storage qualifier that tells the debugger the storage kind of "A" is local storage.
- The debugger uses the storage qualifier to determine how to locate A in device memory

Using TotalView for Parallel Debugging on ANL

TotalView remote debugging on Linux and Mac OS

- Download and install TotalView on your linux or mac.
- Connect to remote front node.
- Run labs remotely



The screenshot shows the 'Preferences' dialog box with the 'Remote Connections' tab selected. The dialog is titled 'Preferences' and has a close button (X) in the top right corner. On the left side, there is a sidebar with icons for 'DISPLAY', 'ACTION POINTS', 'SEARCH PATH', 'PARALLEL', 'REMOTE CONNECTIONS' (highlighted), 'TOOL BAR', and 'LABS'. The main content area is titled 'Remote Connections' and contains the following fields and options:

- Remote Connections**: Create a configuration to launch the TotalView debugger on a remote host. Provide the necessary information to reach the host and the debugger.
- Select a configuration to edit: or [create a new configuration](#)
- Connection Name**: Provide a name for your connection. REQUIRED
- Remote Host(s)**: Provide a comma separated list of SSH hosts in the form: user@host1, user@host2 REQUIRED
- Private Key File**: Specify an optional local private key (e.g. .pem or .ppk) file to use for the SSH connection.
 BROWSE...
- TotalView Remote Installation Directory**: Enter the path to the totalview executable or leave blank if totalview is in your path.
- Remote Command(s)**: Enter optional shell commands to execute or source a remote script before starting remote TotalView debugger
- [Delete Configuration](#)
- Buttons: OK APPLY CANCEL

Hands-on labs

- Install TV from installers on Mac or Linux.
 - Ignore license code
- Star TotalView
- Remotely connect to cooley and enable Reverse Connection

Labs:

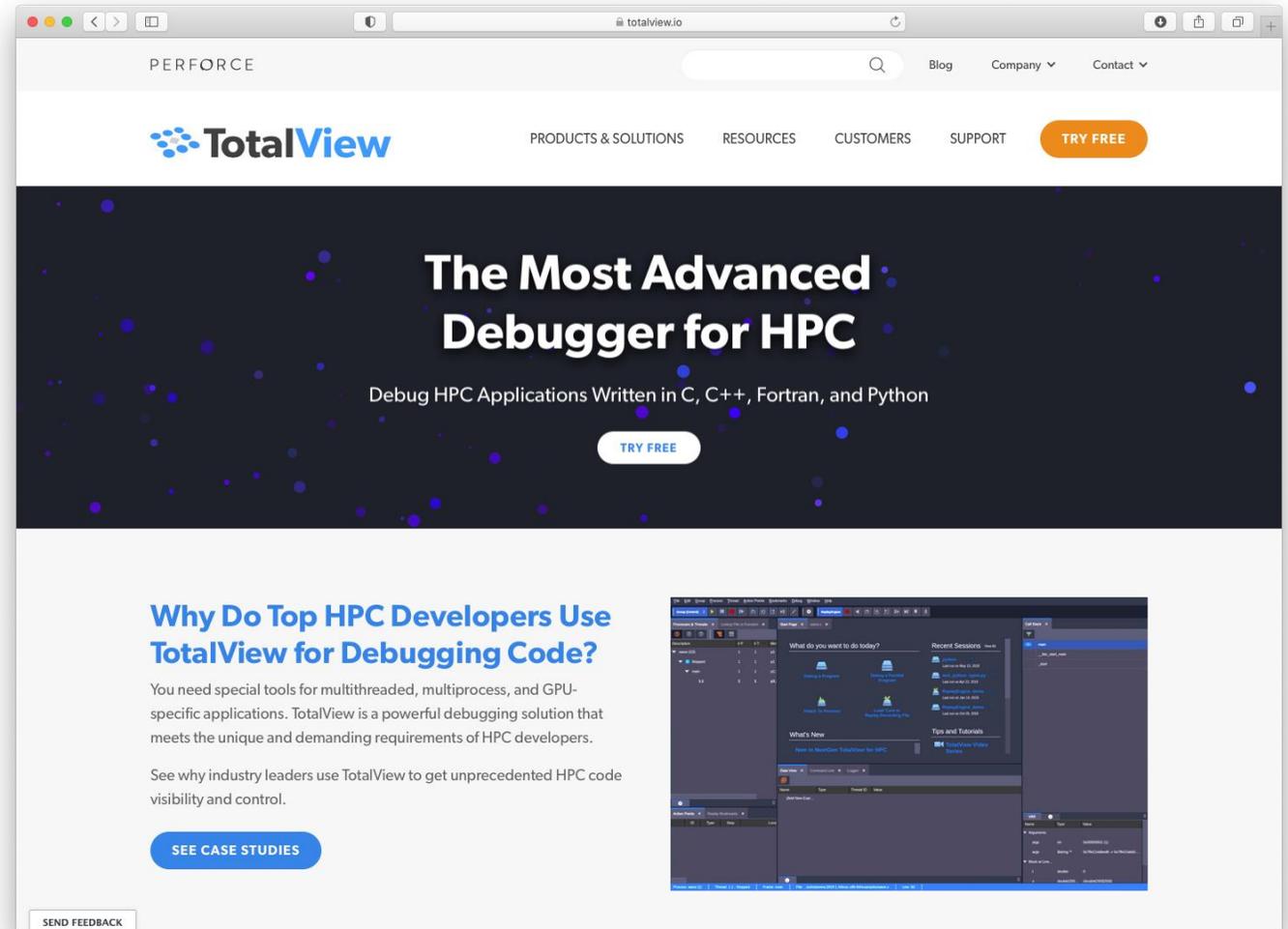
- Lab 1 Debugger Basic
- Lab 2 Viewing, Examining, Watching and Editing Data
- Lab 3 Examining and Controlling a Parallel Application (on Cooley)
 - Using remote connect (tvconnect)
 - `qsub -q training tvconnect.job`
 - Modify and submit `tvconnect.job` on your machine

TotalView is available on Theta, Cooley

- Installed at: `/soft/debuggers/totalview-2021-08-04/toolworks/totalview.2021X.3.756/bin/totalview`
- Connect to Cooley/Theta
 - Get allocation first
 - `qsub -A ATPESC2021 -n 4 -q debug-flat-quad -l (theta)`
 - `qsub -A ATPESC2021 -n 4 -q training -l (Cooley)`
 - `module load totalview (theta)`
 - `soft add +totalview (cooley)`
 - `totalview -args mpiexec -np <N> ./demoMpi_v2`
 - `tvconnect mpiexec -np <N> ./demoMpi_v2`

TotalView Resources and Documentation

- TotalView website:
 - <https://totalview.io>
- TotalView documentation:
 - <https://help.totalview.io>
- TotalView Video Tutorials:
 - <https://totalview.io/support/video-tutorials>
- Other Resources:
 - Blog: <https://totalview.io/blog>



Summary

- Use of modern debugger **saves** you time.
- TotalView can help you because:
 - It's **cross-platform** (the only debugger you ever need)
 - Allow you to debug accelerators (GPU) and CPU in **one session**
 - Allow you to debug **multiple languages** (C++/Python/Fortran)



THANK YOU